TEC
Technik

Hochschule Reutlingen
Reutlingen University

Reutlingen University
Faculty of Engineering
Mechatronics Bachelor

10th February 2020

A Recurrent Neural Net Approach to Activity Recognition

Bachelor's Thesis

Christoph Johannes Jabs

Martin-Vollmer-Weg 19
72144 Dusslingen
Matrikelnummer: 761208

Supervisor: Prof. Dr. Eberhard Binder, Prof. Dr. Christian Höfert

ERKLÄRUNG ZUR EIGENSTÄNDIGKEIT

Ich versichere, dass ich diese Thesis/Arbeit ohne fremde Hilfe selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlichen oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe. Die Arbeit wurde noch keiner Kommission zur Prüfung vorgelegt und verletzt in keiner Weise Rechte Dritter.

Christoph Johannes Jabs

In cooperation with

Sperrvermerk

Die vorliegende wissenschaftliche Arbeit beinhaltet vertrauliche Informationen der Bosch-Gruppe und darf Dritten — außer Mitarbeitern der Hochschule im Rahmen des hoschulinternen Prüfungsverfahrens — nicht zugänglich gemacht werden. Veröffentlichungen und Vervielfältigungen, auch nur auszugsweise, sind ohne ausdrückliche Genehmigung der Bosch-Gruppe untersagt. Ab 5 Jahren nach Abgabedatum unterliegt diese wissenschaftliche Arbeit nicht mehr der Geheimhaltung.

Acknowledgement

I would first like to thank my thesis advisor Prof. Dr. rer. nat. Eberhard Binder of the Faculty TEC at Reutlingen University. He had always constructive criticism through the course of writing my thesis and kept me on track for reaching the goal of this work.

Thanks is also due for Eugen Ruff and especially Jan Eisenberg from the Software Campus of the Robert Bosch GmbH. From them I got not only entrusted with the task for my Bachelor's Thesis, I also got great general advice on scientific working and giving presentations. In addition to that they managed the organisational tasks related to writing the thesis at Bosch.

Lastly, I want to acknowledge Prof. Dr. rer. nat. Christian Höfert, also of the Faculty TEC at Reutlingen University, as the second reader of this thesis.

Christoph Jabs

Reutlingen, 10th February 2020

Abstract

A common approach in current research publication on analysing multimodal time-series data from inertial sensors is to use Convolutional Neural Networks (CNNs) to automatically learn patterns in the given data. The disadvantage when using CNNs in such an application on mobile devices is that they are highly computationally complex. Therefore, this thesis studies the question if Long-Short-Term-Memory (LSTM) networks, which are significantly less computationally complex, can be used as an alternative for such applications.

Firstly the topic of Human Activity Recognition (HAR) and the used datasets Opportunity and mHealth are presented. Following that, inertial sensors and their functional principles are briefly touched on. In the main part of the work, the methods used, mainly LSTMs and CNNs are laid out. In our experiments, we mainly present an architecture study in which 380 LSTM variants were trained and tested on the Opportunity dataset. A modified training process called Relaxed Truth Training that speeds up the training process slightly is also presented and tested in our experiments. When comparing the best two LSTM networks from the architecture study to two different CNN architectures, we found superior performance from the LSTMs, mainly when looking at close to real life, continuous classification problems. These results were verified by testing the same architectures on the mHealth dataset.

Contents

Acronyms **XIII**

1 Introduction **1**

2 Human Activity Recognition **3**
- 2.1 Definition . . . 3
- 2.2 Data Collection and Datasets . . . 4
  - 2.2.1 The Opportunity Dataset . . . 6
  - 2.2.2 The mHealth Dataset . . . 7
- 2.3 Current State of Research . . . 10

3 Sensors for Human Activity Recognition **12**
- 3.1 General Functional Principles for Force Measurement . . . 12
- 3.2 Accelerometer . . . 13
- 3.3 Gyroscope . . . 14
- 3.4 Magnetometer . . . 15
- 3.5 Signals and Quantization . . . 16
- 3.6 Sensors Used in the Experiments . . . 18

4 Methods of Analysis **19**
- 4.1 Feature Extraction . . . 19
  - 4.1.1 Signal Magnitude Area . . . 20
  - 4.1.2 Fourier Transformation . . . 20
  - 4.1.3 Spectral Entropy . . . 21
  - 4.1.4 Feature Normalization . . . 22
  - 4.1.5 Preprocessing in the Experiments . . . 22
- 4.2 Artificial Neural Networks . . . 24
  - 4.2.1 Basic Architecture . . . 24
  - 4.2.2 Training of Neural Networks . . . 27
  - 4.2.3 Convolutional Neural Networks . . . 29
  - 4.2.4 Recurrent Neural Networks . . . 31
  - 4.2.5 Computational Network Complexity . . . 35
  - 4.2.6 Network Performance Measures . . . 37
- 4.3 Neural Networks for Activity Recognition . . . 38
  - 4.3.1 CNNs for HAR . . . 38
  - 4.3.2 LSTM Approach to HAR . . . 40

5 Experiments and Results **44**
- 5.1 Experiments on Opportunity . . . 44
  - 5.1.1 Learning Parameter Study . . . 44
  - 5.1.2 Architecture Study . . . 45
  - 5.1.3 Comparison to CNN Performance . . . 50
  - 5.1.4 Comparison to Other Approaches . . . 53
- 5.2 Validation on mHealth . . . 54

6 Conclusion **57**

Bibliography **XV**

Appendices **XVIII**

A Appendix for Chapter 2 XX
B Appendix for Chapter 4 XXII
C Appendix for Chapter 5 XXXVI

List of Figures

2.1 Example signals of the activity classes standing and walking. 5
2.2 Sensor positions in the Opportunity and the mHealth datasets. 7
2.3 Samples of the gyroscope and magnetometer data in mHealth. 9

3.1 The general construction of a capacitive force sensor. 13
3.2 A schematic illustration of the structure of an accelerometer. 14
3.3 A schematic illustration of the structure of a vibratory gyroscope. 15
3.4 A schematic illustration of the structure of a magnetometer. 16
3.5 Illustration of a continuous signal in comparison to its digital equivalent. 17

4.1 Illustration of the feature extraction process. 23
4.2 The structure of the vector of features. 23
4.3 An exemplary ANN for an example calculation. 24
4.4 Two illustrations of ANNs at different levels of detail. 26
4.5 Illustration of the convolution operation. 30
4.6 Data processing in an exemplary CNN. 31
4.7 Illustrations of an RNN. 32
4.8 Illustration of the signal flow inside a LSTM cell. 34
4.9 The grid arrangement of the input data for the CNN2D architecture. 39
4.10 The layer structure of the CNN2D architecture. 39
4.11 The layer structure of our approach to HAR. 41
4.12 Strict ground truth in comparison to the relaxed ground truth. 43

5.1 Comparison of training progress for two combinations of training parameters. . 45
5.2 F-measure on sequence data for networks with different depth. 47
5.3 F-measure on sequence data over network complexity. 48
5.4 Comparison of the training progress between normal and relaxed truth training. 49
5.5 Comparison of predicted class probabilities and relaxed truth target. 50
5.6 Continuous classification excerpt on the Opportunity dataset. 52
5.7 Continuous classification on the mHealth dataset. 56

List of Tables

2.1 Overview of the two datasets as we used them. 9

5.1 Training parameter values tested in the training parameter study. 44
5.2 An overview of the layer variants trained for each of the basic architectures. . . 46
5.3 Average and standard deviation performance in the architecture study. 47
5.4 Comparison of two LSTMs and two CNNs on the Opportunity dataset. 51
5.5 Performance comparison with entries to the Opportunity challenge. 54
5.6 Comparison of two LSTMs and two CNNs on the mHealth dataset. 55

Acronyms

ADL Activities of Daily Living.
ANN Artificial Neural Network.

CNN Convolutional Neural Network.

FLOP Floating Point Operation.

HAR Human Activity Recognition.

IMU Inertial Measurement Unit.

LSTM Long-Short-Term-Memory.

ReLU Rectified Linear Unit.
RNN Recurrent Neural Network.

SMA Signal Magnitude Area.

1 Introduction

Due to the recent trend of embedding many sensors in consumer devices, even on small and portable ones like a smartphone or a smartwatch, enough data is available to extract information in different areas. One area of information extraction using Machine Learning tools that grew in the last years is the classification of human activities.

In this work, we use Recurrent Neural Networks (RNNs) to gain information about the activity a person is performing through analysing data from body worn inertial sensors. To be precise, we use an extension of common RNNs called Long-Short-Term-Memorys (LSTMs) since they perform better on time-series data.

The main goal of this work is not to develop a new classification technique to be used for Human Activity Recognition (HAR) but to validate an approach that was developed for theft detection of electric bikes. Since there are not many algorithms implemented for detecting the theft of an electric bike, we apply the same approach to the active area of research which is HAR. The goal thereby is to compare results from our method to other methods that were proposed for HAR.

In the age of the internet of things, all devices become more and more connected and more and more data is collected and analysed to gain as much information as possible. One's personal smart phone is becoming the centre point of daily life by gathering the information of all the devices and being the control point for smart home and other applications. This is also true for the sector of mobility with car manufacturers like Tesla allowing the user to monitor and control their car via their smart phone.

As well as for cars, this is also happening for electric bikes, from which data about the last trips can already be viewed on the users smart phone. In the next generation of electric bikes, constant monitoring of the vehicle through the smart phone is planned. Internal market studies showed that eBike customers biggest fear is having their expensive bike stolen. Following that line of thought, potential customers might be hesitant to purchase an eBike for the same reason. To reach out to those customers, Bosch is developing a Machine Learning-based algorithm that detects theft and vandalism on an electric bike from data of an accelerometer, a gyroscope and a magnetometer.

A problem when developing a Machine Learning model is that one never knows if the best approach to the given task was chosen. The only way of finding a better alternative is seeing better results when comparing two approaches. Since there are no other approaches to the task of theft detection from inertial sensor data available, this means that the comparison to other approaches needs to be accomplished differently. In an attempt to validate the LSTM-based approach which Bosch is taking for the development of the theft detection algorithm, this thesis tries to answer the following research question.

> How do LSTMs perform compared to other approaches on time-series data from inertial sensors?

To answer this question we take the LSTM-based approach and apply it to the field of Human Activity Recognition. This is an active field of research with publications of results based on different approaches available for comparison. In addition to that, the classification of human activities is also based on time-series data from inertial sensors. This means that the two applications are similar enough for being able to transfer findings from one of the fields to the other.

To be able to compare our approach, we are working with datasets that are available to the public and are specially recorded with the goal to serve as benchmarking tools. This has the advantage of having many references that worked on the same set of data.

Outline The chapters of this work are structured to guide the reader through the task of activity recognition, so that every step from the data acquisition to the recognition of a certain activity can be understood. We begin in Chapter 2 by defining the task of Human Activity Recognition and we present the datasets which we work with for answering our research question. After also giving an overview of the current state of research, in the next chapter we look at the sensor types used in HAR. In Chapter 4 we then present the methods which we use in this work. There, we start with feature extraction techniques, followed by the basics of Artificial Neural Networks (ANNs). We finish this chapter by explaining how ANNs can be used for the task of HAR. This section does also present our LSTM-based approach. In the last chapter before the conclusion, we lay out what experiments we conducted, and we present the results which we achieved. In addition to that we compare our results to results presented in publications on the same task. Lastly, a conclusion about what we tried to achieve is drawn in Chapter 6.

2 Human Activity Recognition

With all the devices in our every-day life getting more and more connected, many new opportunities to process the data collected by the increasing amount of sensors, arise. Analysing this previously unavailable information might be key to accelerate improving our quality of life. The area of healthcare carries great potential of profiting from the increased amount of data. Much information about the state of a person's health can be gained from their activity level, their posture [30] and their heart rate [3], to only name a few examples. With the increase in smart wearable technology, the preliminaries for tracking a subject's activity profile are now created, which is why research on recognizing human activities based on different data has started to become popular. Besides HAR based on video data, most research focusses on classifying movement of humans based on data from inertial sensor systems, because this is already being integrated into smartphones, smartwatches and other wearable devices.

In this chapter we give an overview of the research field of HAR and why we chose this field to test how feasible it is to apply RNN-based approaches to time series data of inertial sensors. After defining the term Human Activity Recognition we talk about the collection of data to be used as training data for algorithms and present two publicly available datasets, which we used in our experiments. Finally, in the last section we give an overview of the current state of research in the area of HAR.

2.1 Definition

Human Activity Recognition (HAR) in general focusses on detecting and recognizing actions of a human subject to monitor their quality of life and health.[2, 18] This can be done based on different data sources with the two most common ones used being image/video data (e.g. in [31]) and data recorded by inertial sensor systems (e.g. in [13, 23, 32]). This work approaches HAR only based on inertial sensor data, therefore, if not stated otherwise, it can be assumed that we are referring to tasks based on data from inertial sensor systems.

Research on HAR is based on the goal of gathering meaningful data about a person's health. Since the amount of activity that one performs in ones daily life has a great influence on the development of ones personal health, predictions and suggestions can be made if the activity profile of a person is known. With the current trend of using smartphones and smart wearable devices to track and monitor ones personal health, HAR for applications on smartwatches and alike has become an active field of research, not only for independent research groups but also for companies. A slightly different application of HAR is monitoring activities of elderly people. The goal in this application is not only to monitor general health but also to detect unusual disturbances in an activity profile. If events, like e.g. a fall, could be monitored remotely, it would be possible to quickly send help to elderly people who are living alone and are unable to call for assistance on their own.

The kind of activities which are sought to be recognized can vary for different applications, datasets and research groups. Most datasets include labels for *modes of locomotion* (e.g. Standing, Walking, Sitting and Lying [25]) and many datasets add classes for walking up and down stairs to that (e.g. in [2]). However, classes can get more intricate and detailed. The Opportunity dataset for example, does also include labels for high level activities like relaxing or drinking from a cup.

Even if restricting oneself to HAR from inertial sensor data, there are differences in the sensor types, the number of sensors and the sensor positioning that is used when working on HAR tasks. The types of sensors commonly used are accelerometers, gyroscopes and magnetometers. They are described in detail in the next chapter. From these sensor types, the one that research on HAR started with is the accelerometer. Works from the beginning of the century [19, 32] were usually based on data from accelerometers alone, since they were the first of the three sensor types that were being implemented into small devices. With full Inertial Measurement Units (IMUs) finding their way into more and more devices, datasets that include data from gyroscopes and magnetometers were introduced [25, 3] and the increased popularity of smartphones did yield datasets which recorded all of their data with a smartphone [2].

After this overview and definition of HAR, in the next section we outline the data collection process to gain data that can be used for HAR, and present two datasets in detail.

2.2 Data Collection and Datasets

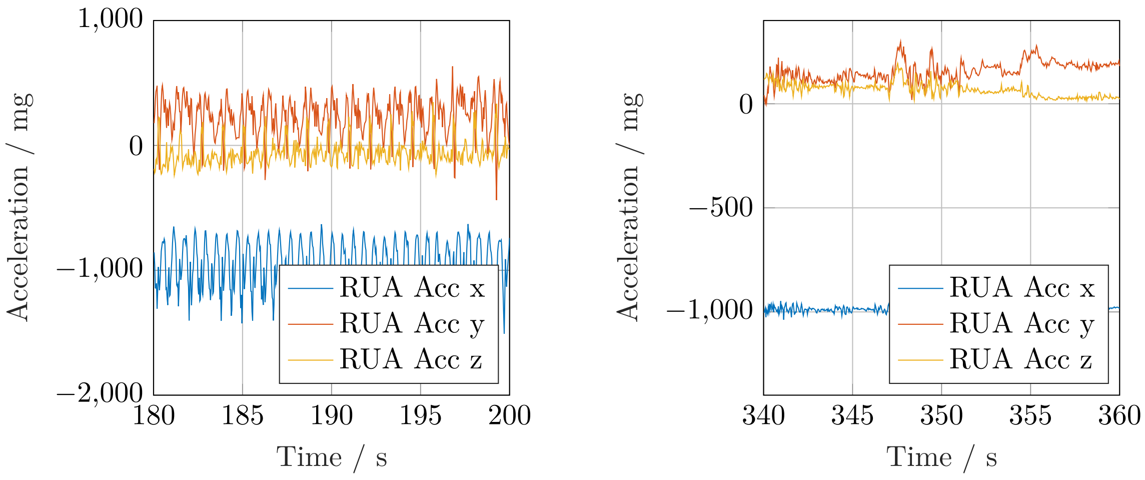
A crucial part of developing a solid Machine Learning model is the data that the model is trained with. Since the data is the foundation of the algorithm, it is necessary to have a dataset which does encapsulate the real-world application reliably and provide enough data samples that overfitting and instability do not impose problems which make it impossible to generate a useful model. To help research and to allow for different approaches to be compared with results on the same dataset, research groups have recorded and made publicly available sets of data which can be used to train models for activity recognition. This section provides an overview of the data collection process needed for recording such a dataset and present the two datasets Opportunity and mHealth in detail.

Recording the data for a HAR dataset comes with many decisions and challenges. There are parameters, like the sensors used, the sensor positioning and the subject selection, where different decisions need to be made, depending on what the dataset should be used for. If a dataset is recorded with the goal of training models that shall be deployed in a smartphone app and recognize human activities based on the data of the IMU of the smartphone, the data for the dataset needs to be recorded with a smartphone to provide training data that best resembles the intended application. Many of the datasets for more general use have been recorded with sensor systems that were developed for research, because they are easier to configure and use for this purpose. An example for such a sensor system for research applications is the Shimmer system [5].

It is also important to think about how the data is being recorded and how to synchronize the data, if it cannot be recorded on one central device. This can be especially challenging if different systems are used in parallel or if the sensors are a long distance apart. In those cases, the signals from different sensors are commonly recorded on different devices and these recordings need to be synchronized afterwards. Since it is difficult to synchronize the recordings automatically, it is likely that the signals need to be aligned in a manual process, like done in [25]. This process is not as accurate as recording all the sensors at one central system, but the accuracy is high enough for the application of activity recognition.

Another process in creating a dataset which usually is done manually is the labelling of the data. The labels form the ground truth for the training process, and they associate each sample of data with one of the activity classes that the dataset contains. Since the labels need to form a reliable ground truth, there is no automated way of creating them from the data. Because of that, most research groups who are recording a dataset opt for capturing the whole recording duration on video and, from that recording, manually label the data afterwards. This is a time-consuming process and does also introduce some variability, since the beginning and the end of an activity can not be defined precisely.

During labelling, a person is watching the video of the subject and decides if what the subject is doing can be classified as one of the activities, which the dataset contains as a label. If that is the case, then that period of time is labelled with the given activity class. If what the subject is doing cannot be described with the classes in the dataset, then the time period is marked as the so-called



(a) A data segment which was recorded while the subject was standing.

(b) A data segment which was recorded while the subject was walking.

Figure 2.1: Example signals of an accelerometer at the subjects right upper arm whilst standing and walking.

Null Class. The Null Class therefore contains samples of data, which are not classifiable with the labels of the dataset. It is important to note, that this does not mean, that the Null Class does not contain any activities or movement, they are just not explicitly labelled.

Two last important aspects, which need to be considered when recording a dataset for HAR, are the environment in which the data is recorded and what the subject is told to do during the recording. Both of these things do influence how naturally the person performing the activities behaves. Since the dataset should resemble the real world as good as possible, it is important that the subject behaves like it normally would and is not restricted by the environment or what it is told to do. This is why many datasets are recorded in a flat or something comparable and not under laboratory conditions. Additionally, most research groups do not tell the subject to walk for two minutes, then sit down for three minutes without any motivation for doing so, but rather opt for specifying a sequence of high level activities (e.g. ‘move in the room, check the objects are in the right places’[25, p. 3]) that the subject is to perform. The low level activities do then arise naturally out of performing the higher level tasks.

To get an idea of how the data in a dataset for HAR looks, Figure 2.1 shows two segments of different activities from a dataset. The signals plotted in the two sub figures are the three axis signals of an accelerometer which is mounted at the right upper arm of a subject. Figure 2.1a shows 20 seconds of the data of said sensor that was recorded while the subject was standing. This part of the data has a continuous label of the standing class in the dataset. The other sub figure shows a segment of the same length and from the same sensor, but the subject was walking during the recording. This segment therefore has labels for the walking class associated with each of the data samples.

When comparing the two signals, it is obvious that the signal of the standing class is significantly more static than the one of the walking class. In the signal shown in Figure 2.1b, a periodicity arising from the frequency of steps taken by the subject can clearly be seen. Based on observations like that, the different activity classes can be distinguished based on only the data of inertial sensors.

After laying out these parameters, which need to be considered when recording a dataset for HAR, we present now the two datasets which we were using in our experiments. These are the Opportunity dataset, which is introduced first, and the mHealth dataset. We did choose these datasets based on [18], which provides a review of different datasets for HAR and served as a good starting point for choosing the datasets which fit our needs.

2.2.1 The Opportunity Dataset

The Opportunity dataset was recorded in a studio-flat like room and does include a total of 72 sensors. In addition to that, it includes seven label tracks, which describe the activities at different levels of abstraction. This makes the dataset a good foundation for Machine Learning experiments on the topic of HAR because it contains much data and allows for only using a subset of the sensors and therefore create a smaller dataset.
All information about the Opportunity dataset can be found in [25].

The creators of the Opportunity dataset defined two different scripts, which they called an Activities of Daily Living (ADL) and a drill run. The ADL run has the subject doing activities which ‘may be summarized as “get up”, “coffee”, “sandwich”, “clean” and “break”.’[25, p. 3] This aims to create a natural sequence of activities which the subject might do on a normal morning. In distinction to that, the drill run includes 20 repetitions of the following rudimentary activities which are cited from [25].

1. Open and close the fridge
2. Open and close the dishwasher
3. Open and close 3 drawers (at different heights)
4. Open and close door 1
5. Open and close door 2
6. Turn on and off the lights
7. Clean table
8. Drink (standing)
9. Drink (sitting)

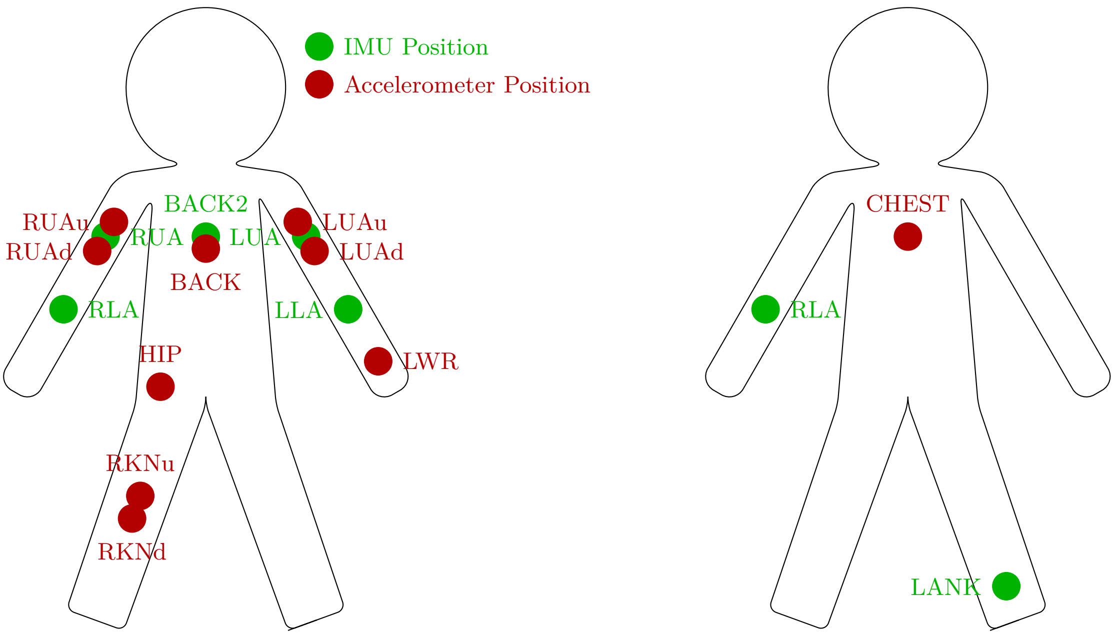
The Opportunity dataset was recorded with twelve subjects, however only the data of four of those subjects is available online. Each of the subjects did perform five ADL and one drill run, which were all recorded as separate sequences. This provides a total amount of about eight hours of data.

Opportunity aims to create a sensor-rich environment in which the ADL and the drill runs were recorded. 72 sensors were not only placed on the body of the subject, but also on objects and furniture in the room, which allows for detecting interactions between the subject and objects. The data of the Opportunity dataset was labelled manually based on video footage that was recorded while the subject did perform the different runs.

For our experiments, we decided to focus on classifying the modes of locomotion, which the dataset provides as its most basic track of labels. The activity classes of modes of locomotion contained in the Opportunity dataset are the following five.

- Standing
- Walking
- Sitting
- Laying
- Null

Since the modes of locomotion do not include any interaction with objects, we decided to use only the sensors, which were attached to the subject’s body. This leaves us with a total of 17 sensor positions where different kinds of accelerometers and IMUs were placed. Out of these 17 positions we had to remove three positions, because more than 95 percent of the data points from that position were missing for at least one of the recordings. This left us with the sensor positions which are marked in Figure 2.2a. At the positions where only the acceleration was recorded, the subject carried a custom wireless accelerometer as described in [24]. There were two additional IMUs (InertialCube3) placed on the shoes of the subject, but we did not use their data because



(a) Opportunity (b) mHealth

Figure 2.2: Sensor positions in the Opportunity and the mHealth datasets.

it was in a different format. This means that the IMU data we used was all recorded by XSense sensor units.

We did mention that we could not use some sensor positions because to many data points were missing. This is due to wireless connection problems during the recording of the dataset. For more details on the problems refer to the original source of the dataset. We did handle these missing values by clipping all the recordings in the beginning and the end, if data was missing there, ignoring sensors where more than 95 percent of the data was missing for at least one recording and interpolating the remaining missing data points linearly. Other approaches to solving this problem of the Opportunity dataset are to repeat the last known value or to use a spline interpolation [6].

We did choose the Opportunity dataset because, based on it, the Opportunity Challenge was organized. The Opportunity Challenge was publicly announced in 2011 and research groups could submit their approaches to solve four different tasks in the realm of HAR, which were all evaluated on the Opportunity dataset. The results of the challenge were published in [6] and form a great comparison for other approaches because they include many techniques that were used to solve the problems. For us, task A of the challenge, which is ‘multimodal activity recognition: modes of locomotion’ provides the results that we compare our approach to.

2.2.2 The mHealth Dataset

The other dataset, which we are using for our experiments is the mHealth dataset, that was recorded related to the mHealthDroid Android framework published in [4] and [3]. MHealthDroid was developed as an open source Android framework to reduce the work needed for creating a health focussed app for Android smartphones. The dataset was collected during the development of an activity recognition app which served as a demonstrator for the mHealthDroid framework.
Information about the mHealth dataset is taken from [3].

For collecting the data in the mHealth dataset, Shimmer2 sensors at three positions marked in Figure 2.2b were used. The sensor at the subject’s chest was recording acceleration, as well as a two-lead electrocardiogram. Both remaining sensors are full IMUs with an accelerometer, a gyroscope and a magnetometer.

The subjects for the mHealth dataset were asked to perform a sequence of twelve activities with either a number of repetitions, or the duration they should perform the activity for, specified. This was recorded with a video camera in addition to the sensor data and later on labelled manually with each activity providing a label and the time in-between activities being labelled as the Null Class. The defined activities were the following and are taken from [3].

- Standing still
- Sitting and relaxing
- Lying down
- Walking
- Climbing/descending stairs
- Waist bends forward
- Frontal elevation of arms
- Knees bending (crouching)
- Cycling
- Jogging
- Running
- Jump front and back

To get labels which are more similar to the labels of the Opportunity dataset, we did decide to condense these classes into broader ones, and we introduced a class called *Other* that contains the classes that are not present in Opportunity. For this, we did keep the first three classes, pooled walking, climbing stairs, running and jogging into a walking class, and labelled all the remaining classes as other. In this way we reduced the classes in mHealth down to the following six.

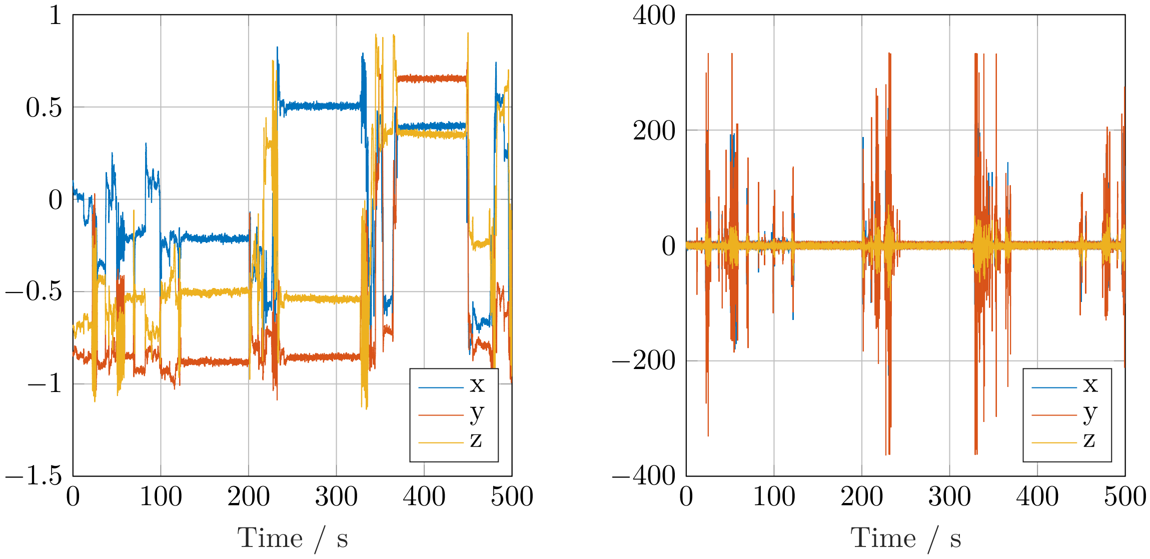
- Standing
- Walking
- Sitting
- Laying
- Other
- Null

The mHealth dataset contains data of ten subjects, performing one run of the defined activities each. This results in a total of almost seven hours of data.

When working with the data of the mHealth dataset, we noticed that the signals for the magnetometer and the gyroscope did look unusual. The signals which were labelled as the gyroscopes data did appear to have a constant signal in times when no movement seemed to be happening according to the acceleration data. In contrast to that, the signals which were labelled as the magnetometer data did always return to zero in those time slots. Samples of the two signals are shown in Figure 2.3. This is unusual because a gyroscope, which measures the rate of turn, should only produce a signal if constant movement is happening whereas a magnetometer does still measure the constant magnetic field of the earth and therefore provides a constant signal if no movement is happening. From these findings, we did figure that there seems to be an error in the labelling of the data, and we did switch the labels for the according signals. This means that we did handle the signal shown in Figure 2.3a as a magnetometer signal, even tough it was labelled to be a gyroscope signal and vice versa for the signal shown in Figure 2.3b.

We did choose to use mHealth in our experiments in addition to the Opportunity dataset to have a second source of data that we could verify our approaches on. This allows us to test if the results that we achieve on the Opportunity dataset can be reproduced on the mHealth dataset.

The most important information about two datasets is summarized in Table 2.1. Additional data plots can be found in the appendices, in Section A.



(a) A section of the data labelled to be gyroscope data.

(b) A section of the data labelled to be magnetometer data.

Figure 2.3: Samples of the gyroscope and magnetometer data of the left ankle sensor in mHealth.

Table 2.1: Overview of the two datasets as we used them.

Features	Datasets	
	Opportunity	*mHealth*
Sample Rate	30Hz	50Hz
Number of Subject	4	10
Length of Data	8 hours	6.75 hours
Number of Used Activity Classes	5	6
Number of Sensorpositions	15	3
Number of Sensors	24	7

2.3 Current State of Research

After we have now presented the datasets we chose for our experiments, in the following we provide an overview of the current state of research on the topic of HAR. This allows the reader to see our work in the context of what was already done by others.

Research on activity recognition was done for more than a decade now. Over that time, different approaches have been applied to the task, which provides us with different Machine Learning techniques to compare our approach to.

The earliest publication on HAR that we could find is [19] from the year of 2001. Even tough this is the oldest publication it does already use Artificial Neural Networks (ANNs) for classifying human activities. Classification of four different activity classes was done on the basis of a multi-layer perceptron, which is a feed-forward architecture. As an input to the network, features which were generated based on principal component analysis and a wavelet transform were used.

Another publication from the first decade of this century is [32]. This publication does also use a classification approach based on a feed-forward network. In contrast to [19] however, they did use a staged classification approach. They trained three different networks. The first network acts as a pre-classifier that discriminates static from dynamic activities. Then the data is fed into one of the other two networks that are trained to either classify the static or the dynamic activities. It is worth noting that [32] does use preprocessing as well, including the Signal Magnitude Area (SMA) and frequency domain features.

There are also approaches to HAR which are not based on ANNs. One of these other approaches was proposed in [26] and is based on Hidden Markov Models. Just like in the last publication which we outlined, [26] does also present a staged classification approach where dynamic and static activities are handled by different models in the second stage. As a preprocessing technique, this publication uses a feature selection based on the importance measure of a Random Forest evaluation.

Another approach to a classification task is the use of Bayesian Networks. In [23], Bayesian and Neural Network architectures were compared on the task of HAR. The ANN, which is a feed-forward architecture, was found to provide good results with less processing time, even tough some Bayesian architectures achieved higher accuracies.

In more recent publications, more complex neural architectures were applied to sensor data for classifying human activities. One of the publications proposing a Convolutional Neural Network (CNN)-based approach is [13]. In this work, the researchers propose a novel CNN architecture which handles acceleration and gyroscope data dissociatively and later fuses them together to form a prediction. This approach is compared to ones based on Hidden Markov Models, Support Vector Machines, Hidden Conditional Random Fields and an approach based on conventional CNNs. The CNN-based approaches were found to outperform the other techniques with the novel approach even outperforming the conventional CNNs in most cases.

The last publication that we want to outline is [16]. Here, the researchers mainly focus on a feature extraction method which they propose, but they are also validating their preprocessing technique by providing results for HAR. Two classification approaches are used, one based on Support Vector Machines and one based on a Radial Basis Function.

Other than these individual publications, we can also look at the approaches that were submitted to the Opportunity Challenge. In [6], results from four approaches of the challenge organizers, as well as eight submissions are shown for the task of classifying modes of locomotion. The approaches of the organizers are based on k-Nearest Neighbour, Nearest Centroid Classifier, Linear and Quadratic Discriminant Analysis, while the submissions add Support Vector Machine- and Decision Tree-based classifiers.

For a deeper understanding of our underlying data, we revisit basic working principals of micro electromechanical sensors in the next chapter.

3 Sensors for Human Activity Recognition

The task of Human Activity Recognition (HAR), which we chose as the field of research to answer our research question in, relies on motion data representing all six degrees of motion. These degrees of freedom are on three translational and three rotational axes. Commonly used sensors to translate any motion into machine-readable format include accelerometers, gyroscopes and magnetometers for their interpretability, simplicity and universality. The combination of three accelerometers and three gyroscopes is well suited to capture movement on the mentioned six degrees of freedom. Therefore, this thesis focuses on these sensor types.

As the first step to modelling is to understand the underlying data, this chapter describes the sensors' working principles, then presents the type of data generated by them and concludes with an overview on the technical specifications of the sensors used for data collection.

3.1 General Functional Principles for Force Measurement

The basic function of a sensor is to convert a physical quantity to an electrical signal, either analogue or digital, for being transmitted to a host device. Although sensors are used to measure a variety of physical quantities, the underlying principle can be summed up into three classes in general. All descriptions in this and the following sections follow the explanations given in [14] and [22].

Piezoelectric Sensors These sensors utilize the effect of piezoelectricity, which is that certain solids like the α-quartz (SiO_2) or lithium niobate ($LiNbO_3$) create a voltage when they are deformed under mechanical stress. This voltage is dependent on the deformation in the solid and, if such a piezoelectric solid is placed in or on a spring in the sensor, can therefore be used to measure the force that the spring exerts. Since the piezoelectric effect is only present when the material is being deformed, a changing force is needed to generate a voltage. If the force is static, no piezoelectric effect is present. Therefore, special measurement circuitry is needed to measure static forces.

Piezoresistive Sensors Sensors based on the piezoresistive effect have a similar functional principle to those based on the piezoelectric effect. The only difference in piezoresistive sensors is that the strain sensitive material does not actively supply a voltage but only change its electric resistance based on the mechanical strain imposed on it. This effect can then be measured by applying an external voltage and arranging multiple piezoresistive elements in a circuit such as a Wheatstone Bridge. With such a circuit, the change in resistance is converted into a voltage signal that is again proportional to the force that acts upon the piezoresistive element.

Capacitive Sensors Information about the amplitude of a force can also be gained by measuring how far a spring is elongated through that force until the force and the reaction of the spring reach an equilibrium. This equilibrium is reached if the force to be measured is equally strong as the force exerted by the spring that can be calculated by the following equation, where k denotes the spring constant and x the elongation.

$$F_{\mathrm{S}} = -k \cdot x \tag{3.1}$$

The elongation can be measured by attaching one or multiple plates or beams to the spring and double that amount of corresponding plates or beams in a fixed position, as shown in Figure 3.1. These three structures are isolated from one another and therefore a capacity C_1 can be measured between one of the fixed structures and the moving one and a capacity C_2 with the other fixed structure respectively. If the spring is elongated through the force that shall be measured, the

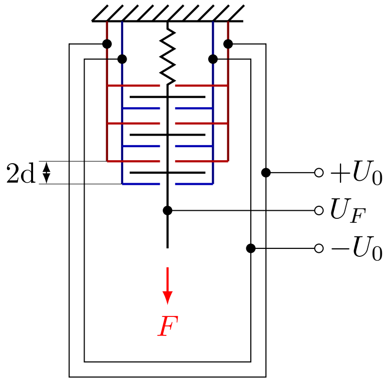


Figure 3.1: The general construction of a capacitive force sensor.

distance between the fixed and the moving structures, and therefore the capacities, changes. From the capacity change one can therefore deduce the force that is acting upon the spring.

In practice, the capacity is not measured directly, but through measuring the voltage U_F, which is explained in the following. Since we are not introducing any new charges into the sensor system during the measurement, we can use the conservation of charges, which states that the charge in the capacitor between the first fixed structure and the charge in the other capacitor need to sum up to zero. If we now apply the voltage $\pm U_0$ to the two fixed structures and use $Q = C \cdot U$ we get the following expression based on the conservation of charges.

$$C_1 \cdot (U_0 - U_F) + C_2 \cdot (-U_0 - U_F) = 0$$

Let d be the distance between the beams of the fixed and the moving structure when no force is applied, then the distance for the two capacitors when force is applied can be denoted as $d \pm x$. If we now use the expression $C = {}^{A\cdot\varepsilon}\!/_{d\pm x}$ and we know that A and ε are equal for both capacitors, we can find the following expression for x depending on U_F.

$$x = -\frac{U_F \cdot d}{U_0} \tag{3.2}$$

Finally, we can combine (3.1) and (3.2) to get the following expression for the force depending on U_F.

$$F = \frac{k \cdot U_F \cdot d}{U_0} \tag{3.3}$$

3.2 Accelerometer

The most common sensor type in HAR is the accelerometer. As the name implies, an accelerometer measures the acceleration that is acting upon a body. Since the acceleration is the second derivative of the distance the object travelled, much information about the objects' movement can be gained from the acceleration signal.

To convert the acceleration to a measurable quantity, a dampened spring-mass-system as shown in Figure 3.2 is used when measuring acceleration on a larger scale or in a micro electromechanical sensor. Newtons' second law of motion, as described by the following equation, shows that an accelerated mass produces a force.

$$F = m \cdot a \tag{3.4}$$

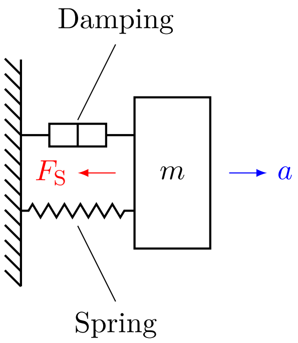


Figure 3.2: A schematic illustration of the structure of an accelerometer.

This force compresses or expands the spring until the force of the spring (see (3.1)) equalizes the inertial force due to acceleration. In this state either the displacement of the mass or the tension in the string can be used as a measure of the acceleration.

In the oldest implementations of accelerometers, the displacement of the mass was measured with the help of a sliding contact. Since this principle is not usable in a miniaturized form factor in modern micro electromechanical sensors, most sensor implementations replace the spring with a capacitive structure as shown in Figure 3.1 and measure the force as explained above. For such a capacitive micro electromechanical accelerometer, the acceleration can be calculated with the following equation, which combines (3.3) and (3.4).

$$a = \frac{k \cdot U_F \cdot d}{U_0 \cdot m} \tag{3.5}$$

Since the vector of acceleration can point in arbitrary directions in space, three individual sensors are needed to capture the direction as well as the amplitude. These three sensors are placed together as one unit with each one of them being turned 90 degrees in relation to the others. The arrangement then outputs three signals that correspond to the acceleration in the directions of the three axes that the sensors are placed on.

3.3 Gyroscope

A gyroscope or gyrometer is ‘a device for measuring angle or angular velocity, based on the gyroscopic effect occurring in rotating or vibrating structures’ [22, p. 5]. Similarly to the described accelerometers most gyroscope sensors are built as a micro electromechanical system.

As said in the previous chapter, the acceleration carries information about the movement of an object through space. Since acceleration is only the change of velocity, an accelerometer does not produce a signal if the velocity is static. This can lead to difficulties when one tries to detect movement with a constant velocity. Gyroscopes on the other hand measure the rate of turn and do therefore not have the problem of giving no signal for constant movement. In contrast to an accelerometer, a gyroscope also measures rotational, not translational movement. Due to these two characteristics, gyroscopes provide additional information about the movement of an object and are therefore used in HAR to gain information about the activity a subject is performing.

A traditional gyroscope is a spinning mass that is mounted in a way so that it can rotate freely around all axes. The law of conservation of angular momentum states that an object spinning around an axis will stay spinning around this axis if no external torque is applied to the object. Due to that, the spinning mass does always stay in the same orientation and can therefore be used as a reference if the external part of the gyroscope is rotated around the spinning inner part.

If the inner part of the gyroscope is not mounted to be able to rotate freely, it exerts a force on the

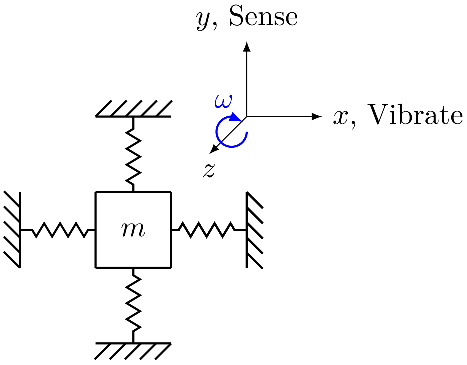


Figure 3.3: A schematic illustration of the structure of a vibratory gyroscope.

outer frame. This force is called the Coriolis Force and can be calculated according to (3.6). It can be measured by one of the methods described in Section 3.1. This is the way most sensors measure the angular velocity that they are rotating at.

$$F_{\mathrm{C}} = -2 \cdot m \cdot \omega \times v \tag{3.6}$$

Since it is difficult to have a spinning and freely mounted part in a sensor, especially in a micro electromechanical one, these sensors usually rely on a slightly different functional principle. The Coriolis Force in micro electromechanical gyroscopes is not caused by the rotation, but rather by the vibration of a proof mass. This effect can be achieved by structures with one or multiple vibrating masses. In the following, we do only provide an explanation for a sing-axis gyroscope based on one mass because this is sufficient for understanding the functionality of gyroscopes in the context of this work. Additional information about micro electromechanical gyroscopes with multiples degrees of freedom, e.g. the tuning fork design which is commonly used, can be found in [9], where the following explanation is also based on.

To create a vibratory gyroscope for one axis, a mass is suspended on springs, so that it can move in two orthogonal directions (x and y). On one of these axes (x), the mass is excited by an external force so that it is vibrating in a resonance mode. If the whole assembly is now rotated around the third orthogonal axis (z), the Coriolis Force acts on the mass in the direction of the second axis (y) and therefore displaces it in that direction. A schematic illustration of such a vibratory gyroscope can be seen in Figure 3.3.
Since the mass is oscillating back and forth on the x-axis, the force on the y-axis, given a constant rate of turn, is also oscillating. By comparing the amplitude and phase of the oscillations on the x- and y-axis, the rate of turn and the direction around the z-axis can be determined. The forces on the y-axis can easily be measured by the described piezo-based measurement principles.

Since the rate of turn is a vectorial quantity, just like the acceleration, it is also measured by three individual sensors that are placed in one unit on three orthogonal axes.

3.4 Magnetometer

A magnetometer or magnetic field sensor is a sensor to measure the magnetic field strength that is present around the sensor. Measuring a magnetic field can be useful either for detecting a magnet in magnetic proximity sensors, or for measuring the magnetic field of the earth to determine the orientation of a device, much like a compass would. This can be useful in HAR for determining the orientation of a subject.

There are various sensor types that can be used for measuring a magnetic field. The most common measuring principles are coils, Hall sensors, fluxgate sensors and magnetorestrictive sensors. All of these measuring principles rely on the physical effect of induction, either directly or indirectly. This

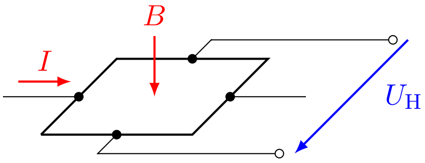


Figure 3.4: A schematic illustration of the structure of a magnetometer.

section focuses on the functional principle of the Hall sensor, since it can be implemented in micro electromechanical sensors and is therefore widely represented in consumer products.

The Hall effect describes that charged particles experience a force, called the Lorentz force, if they travel through a magnetic field. The Lorentz force can be calculated with (3.7) and it can be seen that the Lorentz force is perpendicular to both, the direction of travel of the charged particles and the magnetic field.

$$F_\mathrm{L} = q \cdot (v \times B) \tag{3.7}$$

If travelling through a conductor in a magnetic field, the charged particles are deflected to one side and due to the separation of charges, an electric field is formed. The electric field does also exert a force on the particle and the particles are not deflected any more when the Lorentz force and the electric force given in (3.8) form an equilibrium.

$$F_e = q \cdot E \tag{3.8}$$

The electric field that is present in this equilibrium can be measured on the conductor as a voltage. This voltage is, as is the electric field and the Lorentz force, perpendicular to both, the direction of travel of the charged particles, and the magnetic field and it is dependent on the magnitude of the magnetic field. The structure of a Hall sensor can be seen in Figure 3.4.

The Hall effect is particularly strong in semiconductors and can therefore be used in small sensors by using a plate of semiconducting material as the sensor. A current is driven in one direction through the plate and the voltage can be measured between the other two sides of the plate. This makes the sensor respond to magnetic fields that are perpendicular to the plate itself.

As mentioned for the other two sensor types, magnetometers are also often build up of three individual sensors to be able to measure the direction and amplitude of the magnetic field in three-dimensional space.

3.5 Signals and Quantization

All three sensor types, accelerometers, gyroscopes and magnetometers, measure quantities that change over time. This section describes the steps needed to go from a time-continuous signal, like acceleration, to a time series of data that is saved in a digital format.

The process of recording a signal with a sensor consists of three steps. First the time-continuous physical quantity is converted into an electrical signal. Ideally, this step would be independent of the signal frequency, but in practice it is not. Most sensors do have a limiting cut-off frequency, that is determined by this step. The cut-off frequency in an accelerometer for example is dependent on the proof mass used [14].

Since nearly all data processing today is in digital format, the time-continuous electrical signal needs to be converted into a time-discrete digital one. For this step, an analogue to digital converter is used to sample the time-continuous signal in regular intervals. For each sample the value of the

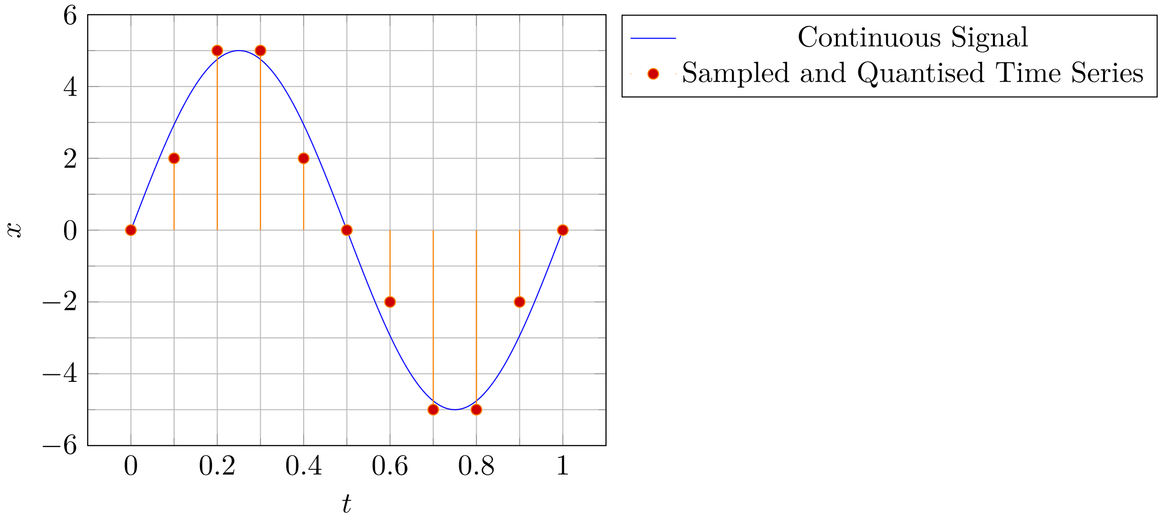


Figure 3.5: Illustration of a continuous signal in comparison to its digital equivalent.

electrical signal is measured and quantized to a digital value that represents the actual value. This process is illustrated in Figure 3.5.

When sampling the signal, the Nyquist-Shannon sampling theorem must be taken into account to prevent aliasing effects. This theorem states that a signal that is limited in bandwidth can always be reconstructed correctly if sampled at a sampling rate that is more than twice as high as the maximum frequency component that is present in the signal [12, p. 236]. If the signal is not limited in bandwidth, it needs to be preprocessed by an anti aliasing low-pass filter to prevent irreversible aliasing effects in the sampled signal.

The last step in recording the signal of a sensor is saving the digital values that the analogue to digital converter generates. If the digital values are saved at the same rate that the converter generates them, the values form a time series of data which represents the original time-continuous signal. Since the rate at which the signal was sampled is known, the signal can be reconstructed from this time series and no information is lost.

Depending on the sensor used, the analogue to digital conversion in the second step is either done by the host device, for a sensor that outputs an analogue signal, or internally in the sensor that then outputs a digital value. If the sampling is done in the sensor, the internal sensor rate is usually significantly higher than the rate at which the sensor outputs the data. This is done as a preventive measure against aliasing effects that could occur if the sensor did internally sample the analogue signal with a lower sample rate.

During the steps from the original signal to the recorded time series, small errors can be introduced at multiple stages. In the conversion to an electrical signal, non-linearities over the signal amplitude and over the frequency as well as effects of hysteresis can be present [22, p. 36]. Some conversion processes do also introduce an offset that should be correct later on [22, p. 38]. The quantization of the signal to a finite set of values does also introduce some quantization errors because each value can only be approximated to a certain degree when having a finite amount of values. This can also be seen in Figure 3.5. It is good to keep these sources of errors in mind, but in most cases the error is small enough that the sensor signals can be used without explicit counter measures.

In addition to these errors, noise is introduced in those two steps. The influence of noise needs to be kept in mind when modelling based on the sensor data.

3.6 Sensors Used in the Experiments

With the basic sensor functionality in mind, we focus on the sensors used in the different data sources. As collecting our own data is beyond the scope of this thesis, we work with the two datasets already presented.

Opportunity If not stated otherwise, all information about the Opportunity dataset is taken from [25].

In the Opportunity dataset, a plethora of sensors is being used. Since this work does only use data from the body worn inertial sensors, only those are listed here.

The subjects in the Opportunity dataset carry 21 inertial sensor units of 4 different kinds. A custom wireless acceleration sensor is the first of these kinds and is described in further detail in [24]. The acceleration sensor in this unit is an ADXL330 and it is used on 12 locations on the body. As another acceleration sensor, two Sun SPOT sensors by the company Oracle are used. The two missing sensor units that are used in Opportunity are Inertial Measurement Units (IMUs), that combine an accelerometer, a gyroscope and a magnetometer. The first IMU, used in a motion jacket on 5 body parts, is a unit by XSense. Since no more information about the exact product by XSense is given, no more information about the sensor can be determined with certainty. The second IMU is a InertialCube3 by the company InterSense. It is used on the two feet of the subjects.

mHealth Information about the mHealth dataset is taken from [3].

In the mHealth dataset, only one sensor platform is used. This sensor platform is Shimmer2 [5]. Acceleration is measures at three positions whilst gyroscope and magnetometer data are additionally recorded at two of these three positions.

4 Methods of Analysis

In modern Computer Science and data analysis there are many tasks that require algorithms for solving them. Developing these algorithms can be a challenging task in itself and becomes increasingly difficult with the complexity of the problem which shall be solved. For problems where patterns need to be detected in excessive amounts of data, developing an algorithm by hand is nearly impossible because the patterns can get to complex for a human to recognize or, if one can notice a pattern, building an algorithm to detect it might not be trivial. This is why methods to automatically find patterns and provide on output based on them were developed. With these methods, algorithms for classifying or predicting a future state from data can be generated if there is enough sample data to train the algorithm with. Because this process is similar to how humans learn new things by being exposed to them, recognizing patterns and then generalizing from them, this process is called *Machine Learning* and has developed to be an important field of research over the last decades.

There are many models that can be used for Machine Learning. Classical forms include decision trees, where a series of binary choices leads to a final prediction, Support Vector Machines, where classes in the data are separated in a multi-dimensional space with the help of hyperplanes, or Bayesian Networks. Whilst many tasks have been solved with the help of these classical models, over the last two decades, most research has been focussed on the topics of Deep Learning and Artificial Neural Networks (ANNs). The increase of interest in these topics has come from their superior ability to solve difficult tasks combined with the increased amount of data and computing power available that was needed for being able to use ANNs effectively.

Since this work does apply Machine Learning approaches based on ANNs to the task of HAR, this chapter provides an overview of what ANNs are, how they generally work and provide explanations of two common architecture types which were used in the experiments explained in Chapter 5, namely RNNs and CNNs. Before we focus on these neural networks, we want to present feature extraction methods, which are commonly used on data before it is fed into an ANN. After that, preprocessing techniques that are used in combination with ANNs are outlined in a separate section.

Our experiments were tested in MATLAB, which is a computing environment by the company MathWorks. Therefore, MATLAB code excerpts for many things that are described in this chapter can be found in the appendices, Section B.2.

4.1 Feature Extraction

As we will see later, ANNs do provide an impressive amount of flexibility to extract information from data, even without processing the raw signals in advance. It can still be useful to do some feature extraction on the data before feeding the signals into an ANN. The advantage thereby is that when choosing the features which should be extracted from the data, domain knowledge can be applied to select the features that provide the most valuable information. Even if an ANN might be able to learn how to extract the same features on its own, this does most likely take a bigger network and more time to train, so when providing the network with features that are already useful, time and computing power can be saved.

In this section we outline features that were used in the experiments of this work. These features were chosen because they extract information from the signals that promise to be useful for classifying human activities. In addition to that, they are the features used in the theft detection algorithm at Bosch and therefore provide additional similarities between the applications if used. In the last part of this section, we explain our preprocessing pipeline from the raw data to the input values used for the ANNs.

4.1.1 Signal Magnitude Area

Common information, which is useful if it can be extracted from a changing signal, is the magnitude of said signal. There are different measures that can be used to represent the size of the signal at a given time. One option would be to calculate the root-mean-square of the signal, another option is the Signal Magnitude Area (SMA).

The SMA was defined in [20, p. 298] as ‘the sum of the areas under the moduli of the integrals, normalized to the length of the signal’. The definition as an equation in [20] was written for three different acceleration sensors. Since we are applying the SMA to the three-channel signals of different triaxial sensors (accelerometers, magnetometers and gyroscopes), we will henceforth write $a_{\{1,2,3\}}$ as $s_{\{x,y,z\}}$. Additionally, we introduce an offset correction to each of the channels, not to make the sequence mean-centred as done in [21], but to start each sequence at a value of zero. For that, the initial values $s_{\{x,y,z\}}(0)$ are subtracted from the signals before adding them. To simplify the equation we are also combining the three integrals defined in [20] into one integral. This leads us to the following definition of the SMA.

$$SMA = \frac{1}{T} \cdot \int_0^T \left[|s_x(t) - s_x(0)| + |s_y(t) - s_y(0)| + |s_z(t) - s_z(0)| \right] \mathrm{d}t \tag{4.1}$$

The SMA was already used as a feature in activity detection and recognition applications because, if applied to the signal of an accelerometer, it becomes relatively easy to distinguish movement from rest by applying a simple threshold. Examples of these applications can be found in [20] and [8].

A common problem in activity recognition is that wearable sensors are not always placed exactly the same way. It is quite common that a sensor might be rotated a bit when comparing the data from one subject to the data of another one. When attempting to build classification algorithms that are not susceptible to such rotational noise, one can resort to features that are inherently rotationally invariant. A rotationally invariant feature is a mathematical expression which computes to the same value, even if the triaxial input data has a rotation added to it. Since the SMA is not invariant to rotation of the input data, we introduced a modification to the definition of the SMA to achieve rotational invariance. By replacing the sum of the channels inside the integral with the euclidean norm, the feature becomes rotationally invariant, because the euclidean distance, which represents the length of the signal vector, is naturally rotationally invariant. Because the SMA is only used as a symbolic quantity and the unit of the SMA is trifling for the usage as an input feature to an ANN, we can simplify the modified SMA by omitting the root of the euclidean norm. This way we get the following definition for our modified SMA.

$$SMA_{\mathrm{mod}} = \frac{1}{T} \cdot \int_0^T \left[|s_x(t) - s_x(0)|^2 + |s_y(t) - s_y(0)|^2 + |s_z(t) - s_z(0)|^2 \right] \mathrm{d}t \tag{4.2}$$

Since we are not working with time-continuous signals, we can not be solving the integral analytically. It is rather solved numerically and in our implementation we opted for an integration based on the trapezoidal method.

4.1.2 Fourier Transformation

Other than from the magnitude of a signal, useful information can be extracted from the frequency of a signal, especially since human activities like walking tend to be periodical. The easiest way of analysing a signal regarding its frequency components is by transforming it into the frequency space. This is done with the help of the Fourier Transformation.

The Fourier Transformation associates a function $x(t)$ with the complex function $X(f)$ of a real frequency variable.[12, p. 173] This is done according to the following relation where i describes the imaginary unit.

$$X(f) = \int_{-\infty}^{\infty} x(t) \cdot \exp(-\mathrm{i}2\pi ft)\mathrm{d}t \tag{4.3}$$

$X(f)$ can thereby be understood as if it is describing with which amplitude and phase a sine wave with the frequency f is present in the original signal $x(t)$. If one wants to know the amount of energy that the signal carries in a frequency spectrum, the energy spectral density can be calculated from the function in the frequency domain. The energy spectral density 'represents the distribution of sequence energy as a function of frequency'[29, p. 5] and is defined by the following equation.

$$S(f) = |X(f)|^2 \tag{4.4}$$

As an input feature for a neural network, different quantities calculated based on the Fourier Transformation can be used. Options for features include the mean frequency of the spectrum, the index of the maximum energy spectral density or the energy in a specified frequency band. More examples of frequency analysis based features can be found in [11] and [10]. In our experiments we were using the option that was mentioned last, based on the following five frequency bands:

1. 0 to 2.5Hz
2. 2.5 to 5Hz
3. 5 to 7.5Hz
4. 7.5 to 10Hz
5. 10 to 15Hz

The energy in these frequency bands is calculated by transforming the signal sequence with the help of a Fast Fourier Transform. After that the energy spectral density of the transformed signal is calculated and normed by the length of the sequence. Since the Fourier Transformation of a real signal produces a spectrum for positive and negative frequencies that is mirrored at the amplitude axis we do only use one half of the spectrum and double its values to get the correct values for the energy spectral density. As a last step, we sum up the energy over the frequency band to get the cumulative energy. This provides us with a single feature value to be used as an input feature for the network which is also rotationally invariant.

4.1.3 Spectral Entropy

Entropy, as proposed by Shannon, is a quantity that 'measures the amount of information generated by the source'[28, p. 396]. The entropy can therefore be used to distinguish noise from a signal that carries information and hence can be useful as a feature for Machine Learning.

Shannon defines entropy in [28] according to the following equation but the constant K can be left out if using it as a feature for Machine Learning since the exact quantity does carry no useful information, only the relative comparison to other entropy values.

$$H = -K \cdot \sum_{i=1}^{N} p_i \cdot \log p_i \tag{4.5}$$

The entropy can be calculated from different signals. One can either apply it directly to the raw time series data of the sensors or apply it to a signal that is already preprocessed. If the entropy is calculated from the probability density function of the spectrum of a signal, one obtains the so-called spectral entropy of the signal. The spectral entropy is a measure of the information contained in the spectral energy distribution of a signal and can therefore effectively be used to distinguish noise from a signal that transmits information. It is calculated by normalizing the energy spectral density, which is determined with (4.4), according to the following equation and treating the output as a probability distribution that the entropy is calculated of.

$$p(f) = \frac{S(f)}{\sum_{k=1}^{N} S(k)} \tag{4.6}$$

In our experiments we calculated the spectral density from the energy spectral densities for each sensor. The values of the spectral density are then used as an input feature for the classification networks.

4.1.4 Feature Normalization

After we did present the different features which were used as inputs for the classification networks in our experiments, we briefly touch on the normalization of features since this is an important aspect when using different features combined.

As outlined in [1], feature normalization is common practice when working with ANNs as well as with other Machine Learning techniques. The most common way of normalizing features is to first centre them around their mean value and then divide them by their standard deviation to get a distribution of the features with a mean of zero and a standard deviation of one. This is done with the following expression where x is all the data points of one feature, μ_x is the mean of that feature and s is the standard deviation.

$$x_{\text{norm}} = \frac{x - \mu_x}{s_x} \tag{4.7}$$

When doing this for all the features, all inputs do have the same distribution of values. The advantage thereof is that no feature is favoured because it has a higher magnitude as another one but all of them have the same influence on the learning.

An important fact to keep in mind when training a network and using only part of the data as the training data is that the values for the mean and the standard deviation should only be calculated from the data used for training and the other data, when used for validation, should be scaled with the same factors. This ensures that the network is only influenced by the training data and that the data for validation is processed in the same way as the training data and therefore provides results as would be yielded in the real world if the data was newly recorded.

4.1.5 Preprocessing in the Experiments

After having described the features which we were using for our experiments, we lay out all steps of preprocessing which we applied to the raw data of the datasets. We calculated the three different feature types for each individual sensor, combining the three axes of the sensors in the process. For the SMA, this combination of the axes can be seen in (4.2), for the power in frequency bands, as well as the spectral entropy, the power spectra of the three axes were summed up.

Since all of these features compute a single value from a sequence of data, we are calculating them based on a window of fixed length, which is then moved one sample at a time. For all the window positions, the seven features are calculated and this therefore generates a sequence of feature values. When thinking about a real-world application, this process resembles a buffer with a fixed size, in which the data of the sensors is stored for some time. Each time a new sample of data comes in from the sensors, the oldest sample is removed and the new one added. This is equal to the shifting of the time window by one sample. The features are then calculated each time for the whole window and the computed values can be fed to the LSTM as a new time-stamp of data. Since we always need at least as many data samples to fill our window or buffer, the feature sequence is $l - 1$ samples

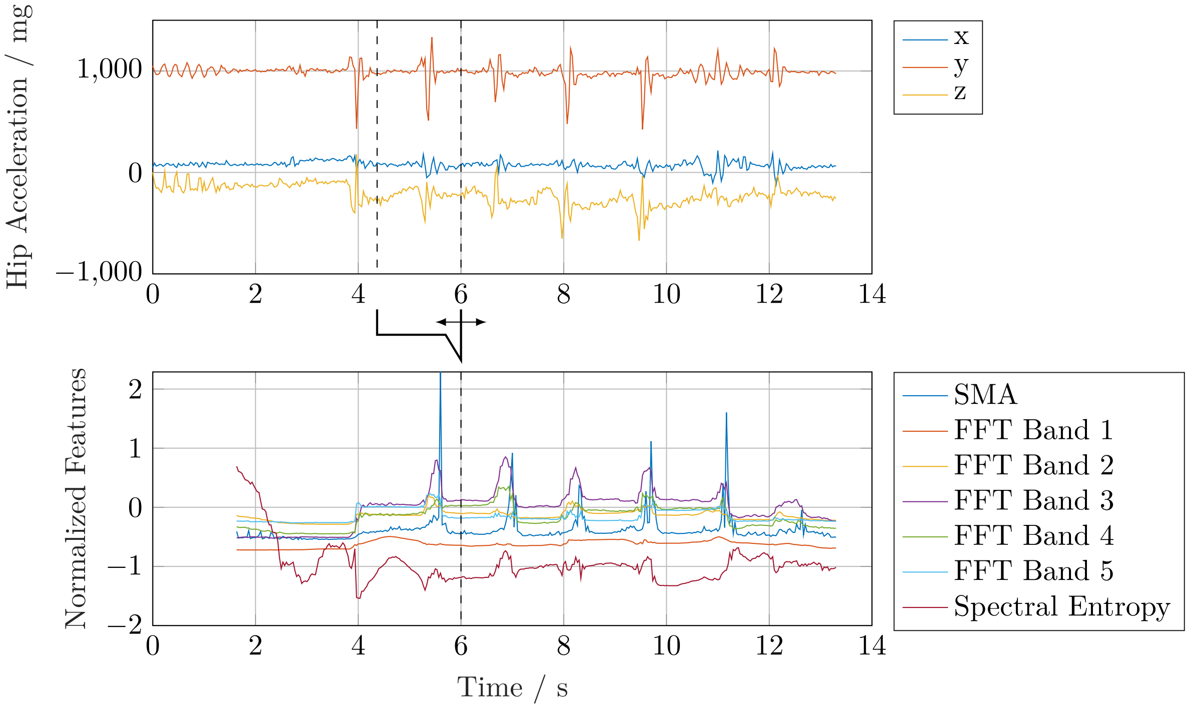


Figure 4.1: Illustration of the feature extraction process.

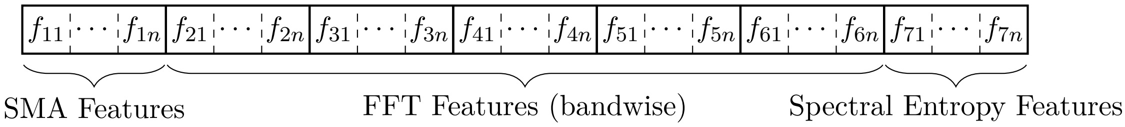


Figure 4.2: The structure of the vector of features.

shorter than the original sequence, if l is the length of the window. We did choose a window size of 50 samples for the feature extraction, therefore we lost a segment of 49 samples at the beginning of each sequence. The windowing process that is used for extracting the feature sequence is depicted in Figure 4.1. In this illustration, the shortening of the feature sequence in comparison to the raw data can also be seen.

After computing the feature values, they are combined into a single vector with all the features of the same type being placed besides each other. For a dataset with n individual sensors, this provides us with a $7n$-dimensional vector for each time-stamp where the features can be calculated. The structure of the feature vector is illustrated in Figure 4.2, where f_{rs} denotes the value of feature r computed from the data of sensor s. Since the two datasets do not have the same number of sensors, the size of the vector of features is different. Opportunity has five positions with IMUs (that each contain three sensors) and nine positions with acceleration sensors for a total of 24 sensors. This results in a feature vector of size 168. The mHealth dataset does only contain two IMUs and one accelerometer, which results in only 49 feature values.

After calculating the features, they are normalized according to (4.7) and are then ready to be fed into the classifying network. Excerpts of the implementation of this preprocessing pipeline can be found in the appendix to this chapter.

In the next section we lay out ANNs with a focus on the two architecture types which are most important for this work, CNNs and RNNs.

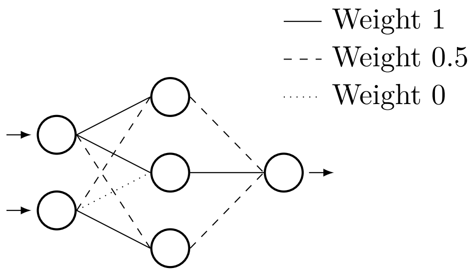


Figure 4.3: An exemplary ANN for an example calculation.

4.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are algorithms which are modelled to resemble the network structure that can be found in the brain of a living being. A brain is build up of many *neurons* that are connected to each other through *axons* and *dendrites*. The points where these axons and dendrites of different neurons meet and make a connection are called *synapses*. The strength of connection in these synapses can vary based on external stimuli.

This architecture is simulated in ANNs. They contain computational units that are, based on their biological inspiration, called neurons or nodes, which are networked through weighted edges. The learning process consists of adjusting the weights in a way to produce better results.
In this section, an overview of ANNs, their general architecture and function as well as two special architectures (Convolutional and Recurrent Neural Networks) that are later used in this work is presented with explanations that are based on the descriptions given in [1].

4.2.1 Basic Architecture

As the name already implies, ANNs are built up as networks. These networks are structured in different layers and can be represented graphically. If we look at an ANN as a black box, we do only see the input and the output layers. Both of them consist of an arbitrary number of nodes where either signals, e.g. the features explained in the previous section, are passed into, or the desired output signals are passed out of the network. In-between the input and the output layer are additional layers which are referred to as hidden layers, because they cannot be seen from the outside.

Before looking at the details of how ANNs work, we want to present an example calculation of a so-called forward pass through the simple network shown in Figure 4.3. This should help in understanding what is happening inside an ANN.

Example (ANN forward pass)**:** As we can see in Figure 4.3, we have two input and a single output value to the ANN. In this example we compute the output value based on the two input values 1 for the top node and -1 for the bottom node. We can write this as an input vector X in the following way.

$$X = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

As we can see, going out of the nodes of the input layer are different connections. Each of them have a weight associated with it. When a signal is passed through a connection, the value of the signal is multiplied with this weight. At each node of the hidden layer in-between the input and the output layer, two connections, one from each input node, come together. What happens at each of the nodes in the hidden layer is that each of the weighted signals are summed up. For the topmost node, this means that the value of the top input node is multiplied by 1 and the value of the lower input node is multiplied by 0.5. If we sum these values up, we get an activation value of $1 \cdot 1 + (-1) \cdot 0.5 = 0.5$. By doing this for all three hidden nodes, we get the following activations for the hidden layer.

$$h = \begin{bmatrix} 0.5 \\ 1 \\ -0.5 \end{bmatrix}$$

The same steps are then repeated for the connections between the hidden layer and the output node. This leads us to the output value Y.

$$Y = 0.5 \cdot 0.5 + 1 \cdot 1 + (-0.5) \cdot 0.5 = 1$$

After this example computation, we generalize the steps we took to compute the output of an ANN. This leads us to a general description of ANNs.

Fully Connected Layers The layers in the exemplary network are so called *fully connected* layers. This name stems from the fact that each node in the previous layer is connected to every node in the current layer. For simplicity, we did leave out the fact, that usually an activation function Φ is applied to all the activation values of a fully connected layer. With this in mind, we can generally describe the fully connected layer with the following equation.

$$h^{(k)} = \Phi(W^{(k)} \cdot h^{(k-1)}) \tag{4.8}$$

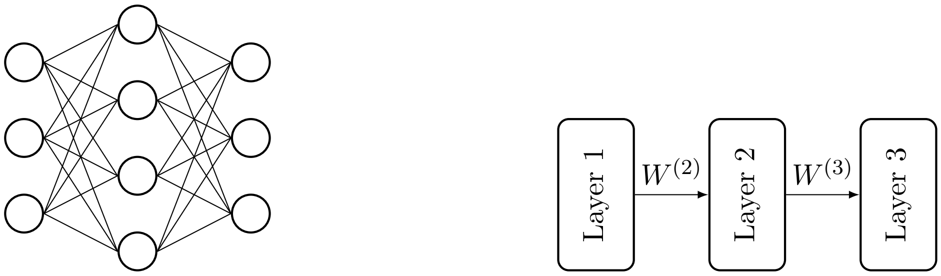
Here the input values to the layer are written as the vector $h^{(k-1)}$ which denotes the activations of the hidden layer before the current layer. In addition, the weights are arranged as a matrix $W^{(k)}$ of size $n^{(k)} \times n^{(k-1)}$ if $n^{(k)}$ denotes the size of the k-th layer. It is important to note that with this notation, the input to the network is denoted as $h^{(0)}$, even tough it is obviously not hidden.

Typical activation functions Φ for ANNs are the hyperbolic tangent, the sigmoid, or the Rectified Linear Unit (ReLU) functions.

For being able to solve problems that include input values that are not zero centred, a *bias* can be introduced to each of the nodes of a layer. The bias is a constant value b which is added to the sum of weighted inputs and can therefore compensate for an offset in the input values. The equation for the fully connected layer given above, with bias is therefore $\Phi(W^{(k)} \cdot h^{(k-1)} + b^{(k)})$. Since the bias can also be modelled as an additional input node that always has the value of 1 and an additional weight, we are not explicitly representing the bias in equations and explanations in the following to provide simpler descriptions.

For creating a deeper neural network, we can now stack as many of these fully connected layers with different sizes as we want. This provides as with other network architectures, as illustrated in Figure 4.4a. Deeper networks have increased abilities to approximate more complex functions and therefore can be used to solve more difficult problems. One does however need more data to train a deeper network, since it contains more weights that need to be adjusted. This is why Deep Learning, which is Machine Learning with Deep Neural Networks has only become popular over the last years, because more data has been collected and can therefore be used to create deep neural models. The stacking of layers is illustrated in Figure 4.4b in a more simplified way that is used in the future of this work. A single drawn connection does hereby represent a multitude of edges in the actual network. A network that does only contain fully connected layers, as illustrated in Figure 4.4, is also referred to as a *feed-forward* network.

Output Layers Depending on what the network shall be used for, there are different output layers that can be used as the last layers of the network. The only function of an output layer is to



(a) The connections of a fully connected hidden layer.

(b) A simplified illustration of the layers in a network.

Figure 4.4: Two illustrations of ANNs at different levels of detail.

convert the output into a form that is more easily interpretable. For example, we might want to interpret the output of each of our output nodes as the probability that the input data belongs to a specific class. This means that the sum of all the output values needs to be fixed at one, because the sum of all probabilities needs to always be one. For this case, the softmax layer can be used. The softmax layer takes in values from a previous layer and converts them into probabilities, depending on how high the activation of the input values are. This is done according to the following equation.

$$h_i^{(k)} = \frac{\exp(h_i^{(k-1)})}{\sum_{j=1}^{n^{(k-1)}} \exp(h_j^{(k-1)})} \quad \forall i \in \{1, \dots, n^{(k-1)}\} \tag{4.9}$$

It is noteworthy that the softmax layer, as well as activation functions to a fully connected layer can be regarded either as an independent layer with the fully connected layer containing no activation function, or to be included in the fully connected layer. In this work we usually consider a fully connected layer to include an activation function, but the MATLAB Deep Learning Toolbox, which we used for implementing our experiments, regards them as two separate things. Therefore, if we talk about a fully connected layer being followed by a softmax layer, it can also be understood as a fully connected layer with a softmax activation.

In the experiments of this work, we used an output layer constellation consisting of a softmax layer followed by a classification layer. Because probabilities are not of much use in a real-world application, the softmax layer is paired with a classification layer in most cases. This layer does take in the probabilities from the softmax layer and output the class with the highest probability. The classification layer is only used when using the finished network, whilst still training, the output of the softmax layer is used.

As a last part that belongs to an ANN there is a *loss function* which is used to calculate the error which the network produced when calculating its output. The loss function therefore takes small values if the network output is close to what it should be and large values if it is a long way off. There are different loss functions that can be used. A few common examples are the squared loss L_{sq}, the loss of logistic regression $L_{\log}$ and the cross entropy loss L_{CE}, which is commonly paired with a softmax output.

$$L_{\text{sq}} = (y - \hat{y})^2 \tag{4.10}$$

$$L_{\log} = \log(1 + \exp(-y \cdot \hat{y})) \tag{4.11}$$

$$L_{\text{CE}} = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i) \tag{4.12}$$

In all three of those equations, $\hat{y}$ denotes the n outputs of the network and y are the outputs that the network should have produced.

Next, we outline the process of training an ANN to understand what is required to get a useful model with the help of Deep Learning.

4.2.2 Training of Neural Networks

Training an ANN is an iterative process which, on a high level, can be described as showing the network examples of data that it should provide an output value for. After that the network is told what output it should have given, for it to adjust its internal structure to provide a better result next time. This is done many times and for many training data points and therefore the network gives better and better results over time.

Gradient Descent In more technical terms, training starts with the loss function defined in the last section. The goal in training is to minimize the loss by adjusting the weights (and possibly other learnable parameters) and therefore improving the output which the network gives. In the training process, the weights of the edges are updated proportionally to their influence on the loss function. This influence is measured by calculating the gradient $\partial L/\partial W$ regarding the weights and updating the weights according to the following equation where α is the learning rate that adjusts how fast the network should learn.

$$W \Leftarrow W - \alpha \cdot \frac{\partial L}{\partial W} \tag{4.13}$$

This process is called *gradient descent* and can be thought of as taking steps straight downhill on the multidimensional loss surface. The goal thereby is to find the point with the lowest loss.

Backpropagation For being able to calculate the gradient of the loss function in a multilayer network, the *backpropagation algorithm* is used. This algorithm computes the gradient of the loss function regarding a weight according to the chain rule of multi variable calculus by adding up all the local-gradient products for the various paths through the network that lead from the output to the weight that the gradient is calculated for. This can be implemented efficiently with the help of dynamic programming by calculating the gradient from the back of the network towards the front and using the previous results in the following calculations, hence the name backpropagation. The chain rule of multi variable calculus, in the context of backpropagation and whilst leaving out some details, looks something like in the following equation. This expression is used to compute the influence of the weights in layer k on the loss of a network with K layers.

$$\frac{\partial L}{\partial W^{(k)}} = \frac{\partial L}{\partial h^{(K)}} \cdot \left[\prod_{i=k}^{K-1} \frac{\partial h^{(i+1)}}{\partial h^{(i)}} \right] \cdot \frac{\partial h^{(k)}}{\partial W^{(k)}} \tag{4.14}$$

Training an ANN therefore has two phases, the *forward* and the *backward* phase. In the forward phase, an instance of the training data is passed through the network to get the corresponding output. This is needed to calculate the loss that the network has for this training instance, which can afterwards be used in the backward phase to calculate the gradient of the loss regarding all the weights in the network. If both phases are complete, the weights can be updated according to (4.13).

Since a typical dataset of training data consists of many samples that can lead to adjust the network in different directions, it is common practice to not update the weights after each instance of training data, but to calculate the gradients of multiple instances and then update the network according to the average of such a *mini-batch*. This is called *mini-batch stochastic gradient descent* because the samples of a mini-batch are randomly selected and provides a good trade-off between stability and speed of the training, with the less stable process being to update the weights after each data sample and the slower process to update over all the training samples at once.

Vanishing and Exploding Gradients A common problem when training ANNs are the so called exploding and vanishing gradients. Since backpropagation uses the chain rule to compute the gradients of layers at the beginning of the network, the backpropagation gives an expression where different gradients are multiplied. The more steps the backpropagation has, the more factors are added to the multiplication. Due to this, the gradients in networks with many layers tend toward zero or extremely high values which results in unstable training because the training either stagnates or progresses in steps that are too big. How severely the problems of exploding and vanishing gradients are depends on many parameters such as the depth of the network or the choice of activation function. By choosing these parameters with the exploding and vanishing gradient problem in mind, their influence can be minimized.
This work does not explicitly talk about steps we took to minimize problems due to exploding or vanishing gradients. The only measure that we talk about is the choice of LSTMs over basic RNNs because they have proven to be less problematic regarding exploding and vanishing gradients. This is explained in more detail in Section 4.2.4.

Overfitting Another common problem in applying ANNs to a task is *overfitting*, where the network performs well on the training data but poorly on data that was not used for training and is therefore not usable in a real-life scenario. This happens when the network is trained for too many iterations on a training dataset that is too small. In this case the network has learned some stochastic patterns in the training data that are only present in this subset of the data. The network is then fitted so closely to the training data that other data has too many differences for the network to be able to perform well on it. This is the main point why training ANNs requires so much data. If a dataset is big, the chance of containing stochastic patterns which do not contain information about the given task reduces greatly, simply because the sample size is bigger.

Penalty-Based Regularization If the size of the training dataset cannot be increased for some reason, overfitting must be minimized with the help of other means. The most common measure to counter overfitting is the introduction of a penalty-based regularization into the network. Regularization is a way of restricting the network without reducing its expressiveness. In the case of penalty-based regularization, this is done by adding a soft penalty for using parameters like weights and biases. There are different penalties that can be used, the most common is L_2-regularisation, which is what the following explanation is using. By increasing the loss proportionally to a factor $\sum_i w_i^2$ the network is penalized for using weights with big absolute values and therefore the network tends to learn patterns with weights that are as small as possible. To implement regularization, the term $\lambda \cdot \sum_i w_i^2$, where λ is a parameter for influencing how strong the network is regularized, is added to the loss. With this addition, the incremental updates of the weights are roughly equivalent to the following equation.

$$W \Leftarrow W(1 - \alpha\lambda) - \alpha \cdot \frac{\partial L}{\partial W} \tag{4.15}$$

What can be seen in (4.15) is that the regularization results in a decay of weights over the training iterations. This means that the weights in the trained network are smaller. In addition to that, the computational complexity of the network is reduced when pairing L_2-regularization with so-called *pruning*. During pruning, after the training all weights which have an absolute value under a certain threshold are set to zero. When a weight value is zero, the corresponding term can be entirely left out of the calculation and the network's complexity is therefore reduced. In this way, parts of the network, which are unimportant for the application, can be removed from the network, without knowing in advance which these parts are.

Dropout Another form of regularization is the use of dropout whilst training. Dropout means that activation values in a hidden layer or input are set to zero with a given probability. This

means that each training iteration only a part of the weights are updated and the other part is not adjusted because they were cut out of the network for that training iteration. By leaving out a random selection of the values in each training iteration, the network is forced to make predictions with a certain amount of redundancy. This adds to the robustness of the prediction and also the ability of the network to generalize from the training data. Since dropout can be seen as adding noise to the training data, overfitting is also reduced.
In MATLAB, dropout is realized as a separate layer, which randomly blocks some of the activations during training. When regarding dropout as a layer, it can be easily shown at which point in the network the values are dropped. Because of this, we are also showing dropout as a layer, when we are using it.

When dealing with overfitting in ANNs it is important to have a reliable way of recognizing if the network has overfit to the training data. For this, the available data is usually divided into three subsets which we refer to as training, test and validation data. The training dataset is the largest of the three and used for performing the training iterations. During the training, the network is regularly tested with the test data and if the test data yields results that deviate significantly from the results on the training data, the training can be stopped because the network has reached overfitting. It is important to note that the network is not updated based on the results on the test data but these results are only used for deciding when to stop training. The validation data is not used during training at all. It can be used to calculate evaluation measures of the finished model and, since the network was not influenced by the validation data at all, it provides a scenario that is as close to a real-life application as possible.

Dropping Learning Rate To increase training stability and to aid convergence, dropping the learning rate is a common technique. When using a dropping learning rate, the initial learning rate α is multiplied with a predefined drop factor in regular intervals T_α, for example every two epochs. An epoch is thereby as many iterations as it takes for all the training data to be fed into the network. With the learning rate drop factor being a value between zero and one, the learning rate gets smaller each time it is multiplied with that factor. Therefore, the adjustments to the parameters of the network are big in the beginning of training and get smaller over time. This aids with reaching a minimum, where too big of a learning rate can change the parameters too much and therefore overshoot the minimum and too small of a learning rate would provide not enough change for the beginning of training. When dropping the learning rate over time, the best of both sides, a big and a small learning rate, can be achieved.

In the following sections two important specialized architectures of ANNs, CNNs and RNNs, is presented. These are the two types of networks that were used in the experiments which are described later on in this work.

4.2.3 Convolutional Neural Networks

An application domain where ANNs did get much attention over the last years is the area of image recognition. Because images encode much of their information in the spatial relationship between different features, Convolutional Neural Networks (CNNs) were developed as a network architecture that analyses data while keeping spatial relations between the data points. This is done by arranging the inputs to the network in a grid structure and searching the data for patterns in both dimensions of this grid.

This section gives a brief overview of how CNNs work. In the experiments we used a CNN as a point of comparing our novel approaches to, which is why a basic understanding of CNNs is required. Since the focus of this work is on Recurrent Neural Networks (RNNs) however, we are not going into much detail.

CNNs were, just like ANNs in general, inspired by research on brains. When Hubel and Wiesel discovered and published that a cat's visual cortex contained groups of cells which were especially sensitive to small regions in the cat's field of view [15], this caused researchers in the field of Machine

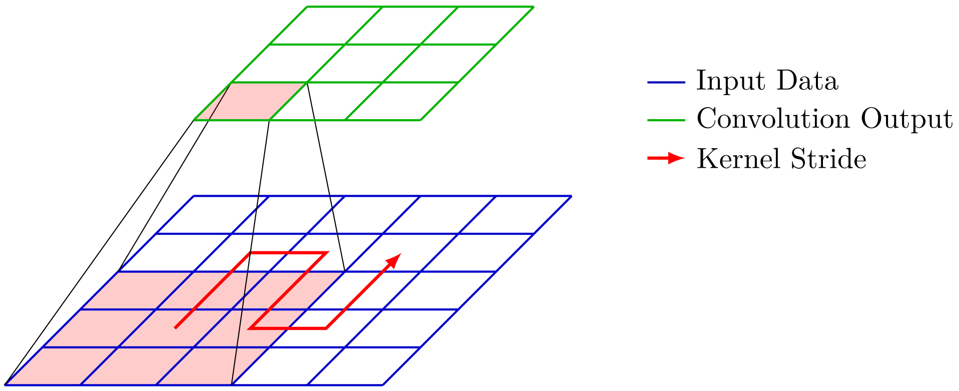


Figure 4.5: Illustration of the convolution operation.

Learning to recreate this in ANNs. What came out of this inspiration was a network architecture that, just as found in the cat's brain, did contain neurons which are closely related to specific regions of the input data. In addition to that, the architecture featured multiple different layers that depict an increasing level of abstraction. The main difference between this original architecture and modern CNNs is that current networks for image recognition rely heavily on weight sharing. This means that a group of cells is not only applied to one part of the image, but is moved over the whole image and can therefore detect a pattern in the data at multiple points.

Convolutional Layers Through most part of the architecture, a CNN is structured as a grid, just like the input data is. A convolutional layer contains a number of so-called *kernels*, which are described later, that are moved over the whole image and generate a value for each point in the image. Hence, each of the kernels produces a new grid of data points which represents the image, but on a higher level of abstraction. All these grids are then combined as a three-dimensional output to the convolutional layer with the grids stacked together as different channels.

Each of the kernels in a convolutional network is a three-dimensional filter with the same amount of channels as the input data and a size for the other two dimensions that can be chosen when planning the architecture of the network. For a CNN that takes a grey-scale image as an input, a common kernel might be of the size $3 \times 3 \times 1$, for a grey-scale image has only one channel of data. This kernel would then have a total of 9 values which are parameters to be learned by the network. During the forward pass through the CNN, in the convolutional layer a *convolution operation* is performed. This operation places the kernel at each possible position over the input data and then calculates the dot product of the kernel and the underlying data, which is illustrated in Figure 4.5. In the following backward pass of training, the values in the kernel are adjusted according to gradient descent.

After the convolution operation, the values produced are usually transformed in some way through an activation function. Theoretically, all kinds of functions can be used for that purpose, but the ReLU has become the standard. The ReLU function is defined as $\max(x, 0)$ and is a relatively new development in the field of ANNs, but because it was shown that the ReLU has great advantages over other activation functions in terms of speed of computation and accuracy of the results, they have quickly gained popularity.

Pooling Layers Because an ANN usually has fewer output than input neurons, the dimensionality of the data needs to be reduced through the network. Since the convolution operations do result in a grid output that is only marginally smaller than the input to the convolutional layer, the size of data in CNNs needs to be reduced differently. One approach is to increase the *stride* of the convolution operation, that is to not move the kernel only one step at a time, but to skip certain points and move it multiple steps at once. This can be done, but might result in certain features not being recognized because the kernel was not placed at a certain position. A more common approach is to include a subsampling or *pooling layer* after the convolutional layer. Pooling layers

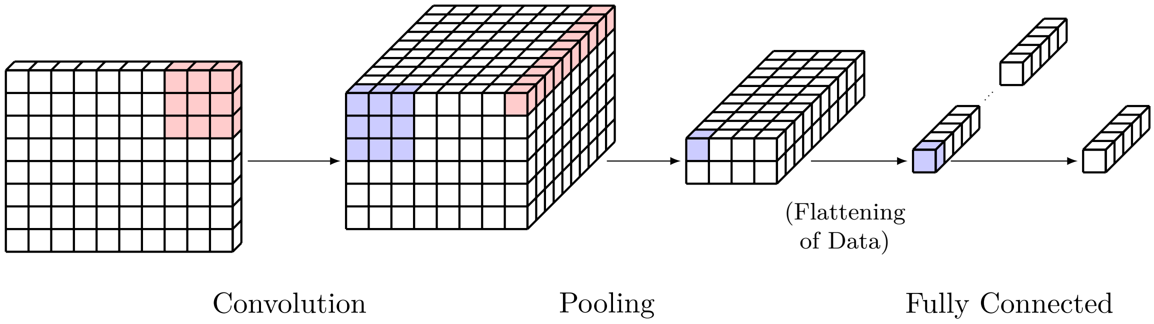


Figure 4.6: Data processing in an exemplary CNN.

work similarly to a convolutional layer in the sense that they do also scan over the data grid in two directions. They do however perform a simpler operation, for example they might only choose the highest value, or calculate the average of the pooling area. When you combine this with a stride that is commonly higher than one, you get a layer that reduces the amount of data, keeps the important features as well as possible, and does not introduce additional learnable parameters.

At the back of a CNN one can commonly find one or more fully connected layers that do compute a final classification or prediction out of the data generated by the convolutional and reduced by the pooling layers. All the data points coming from the last convolutional or pooling layer are fed into the fully connected layers through input neurons and are then treated as independent inputs. Depending on the task, the fully connected layer(s) are followed by an output layer which might for example only use the neuron with the highest activation and interpret it as a classification for a corresponding class. The layer structure of a typical CNN, as well as the dimensions of the input and output data can be seen in Figure 4.6.

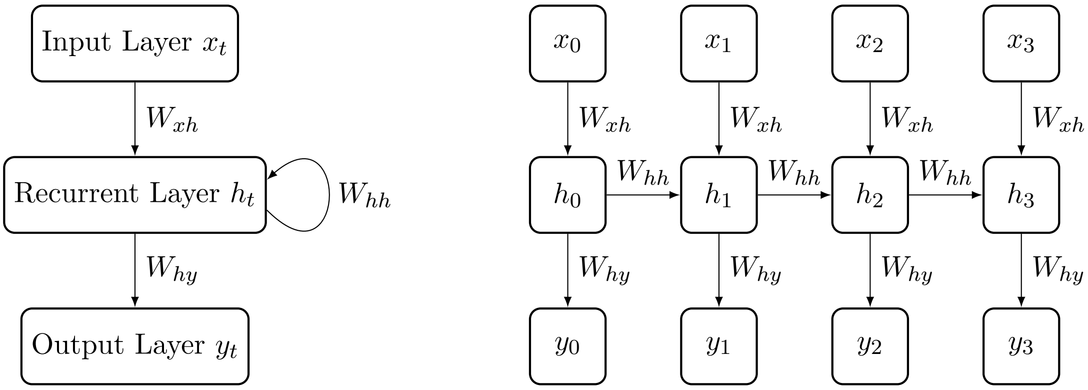
The deepest ANNs used are commonly CNNs. This stems from the fact, which was already mentioned earlier, that each convolutional layer typically increases the level of abstraction. It is quite common to have the first convolutional layer in image recognition detect basic features like edges or points, the next layer would then combine these features to more complex patterns like shapes which can afterwards be combined again by the next layer to form an object that should be detected. The fully connected layers that are following the convolutional layers are then further increasing the depth, but also the expressiveness of the network. Because of the amount of data and the depth of the networks, image recognition is one of the tasks where the most processing power is needed for being able to train networks in a reasonable amount of time.

Up until now we did only talk about CNNs in the realm of image recognition. They can however also be used for other tasks where connections in data should be searched in more than one dimension. An example of such a task is the analysis of data from a multimodal sensor system. In that case there might be patterns in the time dimension of the data, but the sensors might also be related to one another if they are measuring the same action, for example from different points on an object. This is the use-case that we used CNNs in our experiments for.

4.2.4 Recurrent Neural Networks

There are different approaches that can be taken when analysing time-series data with the help of ANNs. For most network architectures the time series needs to be cut into time windows of a fixed length and these windows can then be fed into a feed-forward network or into a CNN as was already outlined briefly. If one however wants to analyse time-series data and keep the context of the entire sequence or analyse sequences of different length with one and the same network, one needs to be using a different architecture. An architecture that can be used for such sequences of different length is the Recurrent Neural Network (RNN) which are presented in this section.

The main advantages of using a RNN over a feed-forward network or a CNN is that it has a fixed



(a) A RNN with one hidden layer.

(b) Time-layered representation of (a).

Figure 4.7: Illustrations of an RNN.

number of learnable parameters which is independent of the sequence length and that it can handle sequences of different length without changing its architecture. This is achieved by feeding the data into the network sequentially, time-stamp by time-stamp. In a normal ANN this would result in independent classifications for each time-stamp, but the hidden layers of a RNN contain a self-loop, as it can be seen in Figure 4.7a, meaning the state of the layer when processing the first time-stamp is fed into the layer when handling the second time-stamp as an additional input. In this hidden state from the last time-stamp, information about the whole history of the sequence can be encoded and all the data points of the past are incorporated when making the prediction for the current time-stamp. To be able to illustrate RNNs easier, they are frequently depicted in an unfolded way, where the different steps through time are represented as layers as they would be in a feed-forward network. This depiction is shown in Figure 4.7b. Note that the weights are still shared between the different time-layers because they are no real distinct layers but much rather the same layer at different points in time.

For a basic RNN containing one hidden layer with $n^{(h)}$ hidden units, let h_t be the $n^{(h)}$-dimensional vector of the activation values of the hidden units at time t. Now let x_t be the $n^{(x)}$-dimensional vector of inputs at time t and y_t respectively the $n^{(y)}$-dimensional output vector. If W_{xh}, W_{hh} and W_{hy} are now the matrices of weights between the input and the hidden units (of size $n^{(h)} \times n^{(x)}$), the hidden units h_{t-1} of the last time step and the current hidden units (of size $n^{(h)} \times n^{(h)}$) as well as the hidden units and the output (of size $n^{(y)} \times n^{(h)}$), the RNN can be described by the following two equations.

$$h_t = \Phi(W_{xh}x_t + W_{hh}h_{t-1}) \tag{4.16}$$

$$y_t = W_{hy}h_t \tag{4.17}$$

Here (4.16) specifies the law for calculating the hidden state from the input and the hidden state at the last time-stamp and (4.17) describes how the output of the network at a specified time is generated from the hidden state. The function Φ, which is applied to all the values of the hidden state is an arbitrary activation function, but commonly in RNNs the hyperbolic tangent function is used. In the recursive nature of (4.16), the ability to use sequences of different length as an input to RNNs can be seen. If the input is longer or shorter than the previous input sequence, the equation is only evaluated more or less often.

Up until now we did only consider RNNs with one hidden layer, but as with the other types of networks it is possible to stack recurrent layers for the network to gain expressiveness. This can be represented in the equations by denoting the hidden state of layer k at time-stamp t as $h_t^{(k)}$. If now the input data is denoted as $h_t^{(0)}$, even if it is not a hidden layer, (4.16) can be generalized to the following equation.

$$h_t^{(k)} = \Phi\, W^{(k)} \begin{bmatrix} h_t^{(k-1)} \\ h_{t-1}^{(k)} \end{bmatrix} \tag{4.18}$$

Here $W^{(k)}$ represents the combined weight matrix of layer k which is formed by stacking the matrices that were formerly denoted by W_{xh} and W_{hh}. Let $n^{(k)}$ be the number of hidden units in layer k, then $W^{(k)}$ has a size of $n^{(k)} \times (n^{(k-1)} + n^{(k)})$. This is done to be able to write the equation in a shorter way and have only one weight matrix that belongs to the input equation of the recurrent layer.
In addition to stacking recurrent layers, fully connected layers can be added at the back of a RNN to work on the outputs of the recurrent layers, just as it is common in CNNs.

Backpropagation Through Time Because a RNN encapsulates the component of time and the weights in the hidden layers are shared between the different time layers, updating the weights in training is slightly different from in a feed-forward network. The main difference is that the backpropagation needs not only to traverse through the layers at one time step, but also through the time layers until the beginning of the network, both spatially and time-wise, is reached. This process is called backpropagation through time and describes how the gradients regarding the weights are calculated during training of a RNN. The law for updating the weights does not change for RNNs and can be seen in (4.13). A detailed mathematical look at the backpropagation through time can be found in the appendix to this chapter.

Due to the fact that RNNs can get exceptionally deep when unfolding them over time, they are especially hard to train. The most common issues in training RNNs are problems regarding vanishing or exploding gradients. Because of the weight sharing between the different temporal layers, the gradients in RNNs tend to contain terms where the same weights are multiplied numerous times and therefore have an even stronger tendency to explode or vanish. A network architecture that was derived from the RNN and does address this problem is the Long-Short-Term-Memory (LSTM).

Long-Short-Term-Memory The conceptual idea behind LSTMs is that RNNs, due to the gradients tending towards increasingly large or small values, are only suited for working with short sequences. One can therefore say that RNNs do provide a good short-term memory, but are not able to store information from long sequences. The LSTM improves upon its parent architecture, the RNN, by adding this missing long-term memory and parameters to have a fine grain control about what data is written to that storage. Adding long-term memory is done by introducing a new vector for each hidden layer which has the same size ($n^{(k)}$) as the vector $h_t^{(k)}$ representing the hidden states. This vector is referred to as the *cell state* and shall be denoted as $c_t^{(k)}$ for the cell state at time-stamp t.

The cell state in a LSTM is controlled through additional parameters which control the ‘learning’ and ‘forgetting’ of the long-term memory. Whilst the weight matrix in layer k with $n^{(k)}$ hidden units in a RNN has a weight matrix $W^{(k)}$ of size $n^{(k)} \times (n^{(k-1)} + n^{(k)})$, the weights of the same layer in a LSTM have the size of $4n^{(k)} \times (n^{(k-1)} + n^{(k)})$. The additional parameters in this weight matrix are controlling the learning and forgetting of the cell states in the layer. Equation (4.18) for a LSTM can be updated to the following system of equations.

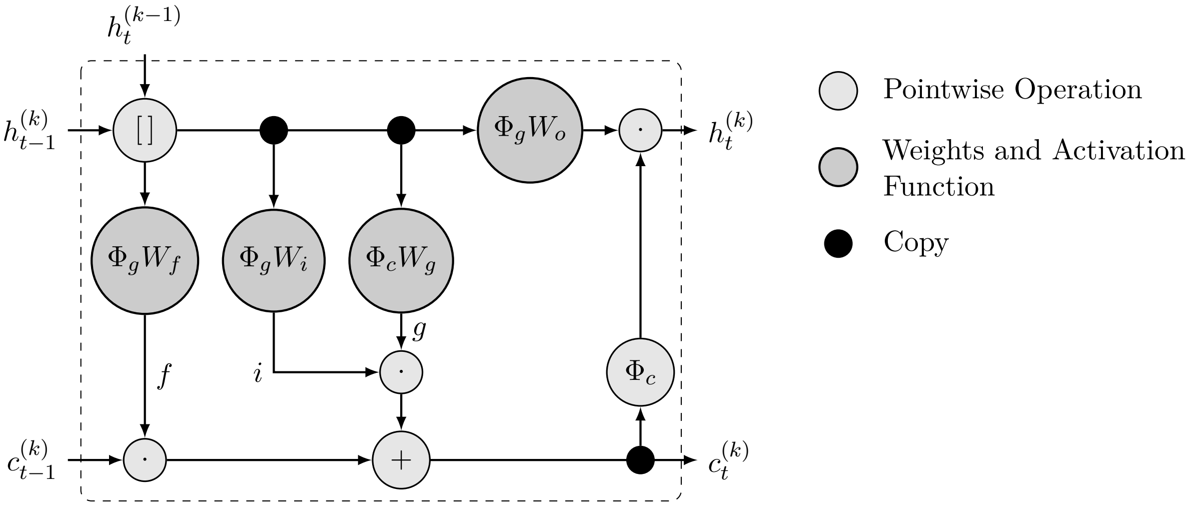


Figure 4.8: Illustration of the signal flow inside a LSTM cell.

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ g_t \end{bmatrix} = \begin{pmatrix} \Phi_g \\ \Phi_g \\ \Phi_g \\ \Phi_c \end{pmatrix} W^{(k)} \begin{bmatrix} h_t^{(k-1)} \\ h_{t-1}^{(k)} \end{bmatrix} \tag{4.19}$$

$$c_t^{(k)} = f_t \odot c_{t-1}^{(k)} + i_t \odot g_t \tag{4.20}$$

$$h_t^{(k)} = o_t \odot \Phi_c(c_t^{(k)}) \tag{4.21}$$

In this equation system, the element-wise multiplication of two vectors is denoted by $\odot$ and Φ_g and Φ_c represent the gate and cell activation functions that are applied in an element-wise fashion. In the equation system, (4.19) describes the first step of updating a LSTM layer in which the *input gate* i_t, the *forget gate* f_t, the *output gate* o_t and the *cell candidate* g_t are computed. Here the gate activation functions typically is the sigmoid function, whilst the cell activation function is usually the hyperbolic tangent. The second equation in the system can be described as selectively forgetting and adding to the long-term memory whilst the third equation describes how the cell state is selectively leaked into the hidden state of the layer.

The gate variables can be interpreted as selecting how much of the proposed cell state is added to the actual cell state (i_t), how much of the old cell state is persisted (f_t) and how much of the cell state is bled through to the hidden state (o_t). When looking at the cell candidate g_t, it can be interpreted as the proposed next cell state. As is can be seen in (4.20), this candidate is passed through the input gate and added to the old cell state, which is passed through the forget gate. The signal flow when updating the LSTM cell in the described way is illustrated in Figure 4.8.

LSTMs set out to solve the problems of vanishing and exploding gradients in vanilla RNNs. They achieve this by removing the derivative of the hyperbolic tangent function and the recurrent multiplication of static weight values in the gradient. The LSTM architecture allows to reduce the cell state gradient to the value of the forget gate. As the forget gate changes its values for different time stamps, this reduces the problems of exploding and vanishing gradients tremendously. A detailed comparison of the gradients of the RNN and the LSTM can be found in the appendix refapp:gradients. There we show mathematically why the LSTM has fewer problems with exploding and vanishing gradients. This is the biggest leverage of LSTMs over RNNs because it improves training stability significantly. Due to that reason we used LSTMs as the architecture of choice in our experiments.

RNNs, and LSTMs alike, can be easily applied to multimodal sensor data by using each channel of the sensors as an input and feeding the time series of data iteratively into the network on a time-stamp basis. The RNNs can then extract information from the relationship of the sensor signals to one another and from the signal progression over time. This approach to HAR is outlined later on, when we present the network architecture which we used in our experiments.

4.2.5 Computational Network Complexity

For being able to compare different ANNs later on, we want to have a measure for the computational complexity of them. Since the computational complexity of a network determines how much time a forward pass through the network is going to take, the goal is to obtain a network which provides the best accuracy with the lowest complexity.

The computational complexity can be estimated by the number of Floating Point Operations (FLOPs) it takes to compute the output of the network. In [33], equations for calculating an estimate of the FLOPs in a feed-forward and a LSTM network are given. From these equations we can extract the following terms for an individual fully connected and a LSTM layer.

$$N_{\text{op}}^{(\text{FC},k)} = 2 \cdot n^{(k-1)} \cdot n^{(k)} \tag{4.22}$$

$$N_{\text{op}}^{(\text{LSTM},k)} = 8 \cdot (n^{(k-1)} + n^{(k)}) \cdot n^{(k)} + 4 \cdot n^{(k)} \tag{4.23}$$

With these equations, the number of estimated FLOPs in layer k of the network can be calculated. The number of nodes in layer k is thereby denoted as $n^{(k)}$. Since our approach is based on networks that contain LSTM, as well as fully connected layers, we can combine (4.22) and (4.23) to form the following equation which can be used for the network architecture which we present later on as our approach to HAR.

$$\begin{aligned} N_{\text{op}}^{(\text{LSTM})} = &\sum_{k \in \mathcal{K}_{\text{LSTM}}} 8 \cdot (n^{(k-1)} + n^{(k)}) \cdot n^{(k)} + 4 \cdot n^{(k)} \\ &+ \sum_{k \in \mathcal{K}_{\text{FC}}} 2 \cdot n^{(k-1)} \cdot n^{(k)} \end{aligned} \tag{4.24}$$

Here $\mathcal{K}_{\text{type}}$ denotes the set of indices of layers of the given type.

In addition to calculating the complexity of LSTM networks, we are also comparing them against CNNs. Therefore, we need an equivalent expression to calculate the computational complexity of a CNN. The following equation for calculating the complexity of a CNN with K layers and two following fully connected layers is also taken from [33] and slightly simplified by removing the term for batch normalization layers, since the CNNs which we are using to compare our approaches against are not using batch normalization layers.

$$\begin{aligned} N_{\text{op}}^{(\text{CNN})} = &\sum_{k \in \mathcal{K}_{\text{conv}}} 2 \cdot c^{(k-1)} \cdot c^{(k)} \cdot n_{\text{kernel}}^{(k)} \cdot n^{(k)} \\ &+ \sum_{k \in \mathcal{K}_{\text{pool}}} c^{(k)} \cdot n^{(k)} \cdot (n_{\text{pool}}^{(k)} - 1) \\ &+ 2 \cdot c^{(K)} \cdot n^{(K)} \cdot n^{(\text{FC1})} + 2 \cdot n^{(\text{FC1})} \cdot n^{(\text{FC2})} \end{aligned} \tag{4.25}$$

Here $n_{\text{kernel}}^{(k)}$ denotes the area of the kernel of layer k, $n_{\text{pool}}^{(k)}$ denotes the pooling area of layer k and

$c^{(k)}$ expresses the number of filters used in layer k. Based on that equation, we did calculate the estimated computational complexity for the CNNs that we compared our LSTMs to. It is important to note that both, (4.24) and (4.25) provide the complexity for one single classification. As we will see later, usually a LSTM is evaluated at each time stamp, whilst a CNN is not processed as often. This means that the complexity values calculated from the two equations can not be compared directly, but the frequency of evaluation needs to be considered as well.

In the following, we give a short exemplary complexity calculation for a LSTM and a CNN to provide a rough estimate of how they compare in that regard.

Example (Complexity Comparison of LSTM and CNN)**:** Consider a LSTM and a CNN, both applied to a sequence of 21 features. The LSTM is evaluated for each time stamp, whilst the CNN is evaluated for windows of 50 samples with an overlap of 50 percent.

Let the LSTM be build up of two LSTM layers of sizes 15 and 10, followed by two fully connected layers with sizes 5 and 3. This leaves us with the following parameters for the LSTM.

$$\begin{aligned}
\mathcal{K}_{\text{LSTM}} &= \{1,2\}\\
\mathcal{K}_{\text{FC}} &= \{3,4\}\\
n^{(0)} &= 21\\
n^{(1)} &= 15\\
n^{(2)} &= 10\\
n^{(3)} &= 5\\
n^{(4)} &= 3
\end{aligned}$$

Let the CNN on the other hand be comprised of one convolutional layer with a kernel of size 3 × 3, depth 20 and stride 1. Following that is a pooling layer with pooling area 3 × 3 and stride 2 in both dimensions. The CNN is finished with two fully connected layers of sizes 5 and 3. Therefore, the parameters of the CNN are the following:

$$\begin{aligned}
\mathcal{K}_{\text{conv}} &= \{1\} & & & & \\
\mathcal{K}_{\text{pool}} &= \{2\} & & & & \\
K &= 2 & & & & \\
n^{(0)} &= 21 \cdot 50 & c^{(0)} &= 1 & & \\
n^{(1)} &= 19 \cdot 23 & c^{(1)} &= 20 & n^{(1)}_{\text{kernel}} &= 3 \times 3\\
n^{(2)} &= 9 \cdot 11 & c^{(2)} &= 20 & n^{(2)}_{\text{pool}} &= 3 \times 3\\
n^{(FC1)} &= 5 & & & & \\
n^{(FC2)} &= 3 & & & &
\end{aligned}$$

With the help of (4.24) and (4.25) we can now calculate the complexities of the two networks.

$$\begin{aligned}
N_{\text{op}}^{(\text{LSTM})} &= 8 \cdot (n^{(0)} + n^{(1)}) \cdot n^{(1)} + 4 \cdot n^{(1)} \\
&\quad + 8 \cdot (n^{(1)} + n^{(2)}) \cdot n^{(2)} + 4 \cdot n^{(2)} \\
&\quad + 2 \cdot n^{(2)} \cdot n^{(3)} \\
&\quad + 2 \cdot n^{(3)} \cdot n^{(4)} \\
&= 6550 \text{FLOP} \\
N_{\text{op}}^{(\text{CNN})} &= 2 \cdot c^{(0)} \cdot c^{(1)} \cdot n_{\text{kernel}}^{(1)} \cdot n^{(1)} \\
&\quad + c^{(2)} \cdot n^{(2)} \cdot (n_{\text{pool}}^{(2)} - 1) \\
&\quad + 2 \cdot c^{(2)} \cdot n^{(2)} \cdot n^{(FC1)} + 2 \cdot n^{(FC1)} \cdot n^{(FC2)} \\
&= 275790 \text{FLOP}
\end{aligned}$$

Even if we now consider that the LSTM is executed 25 times as often as the CNN, we do still have a factor of $N_{\text{op}}^{(CNN)}/25 \cdot N_{\text{op}}^{(\text{LSTM})} \approx 1.7$ that the CNN is more computationally complex.

As stated, these equations do only provide an estimate of the FLOPs required to evaluate the given layers. This stems from the fact that the equations do 'ignore the computation cost associated with non-linear activation functions as it is negligible in comparison to the number of FLOPs in the linear operations.'[33, p. 2].

4.2.6 Network Performance Measures

As a second criterion to compare different networks at, we need to be able to compute a measure of performance from the results produced by them. Four such measures, which we used to compare the networks in our experiments, are introduced in this section.

The first distinction which needs to be made is what data the performance measure is based on. We can either calculate the performance based on segments of the validation data which are prepared in the same way as the training data is. This preparation process of the training data is explained in detail when we present our LSTM approach to activity recognition. For now, it shall only be noted that this process produces data segments with a fixed length and only a single activity class in them. The two performance measures based on this segmented data are referred to as *segment* measures and the symbols in formulae get the abbreviation *seg* as an index. As another kind of data, we can also compute the performance measures based on feeding the entire validation sequences continuously into the network. We call all measures based on the performance for this classification method *sequence* measures and append the index *seq* to their denoting symbols.

Classification Accuracy The first of the two measure types we use is the *classification accuracy*. When talking about classification tasks, the accuracy is defined as the number of the correct predictions divided by the complete number of labels. For a sequence classification as done by LSTMs, this means that the accuracy is calculated for the classifications at each time stamp. This can be expressed as the following equation.

$$a = \frac{N_{\text{correct}}}{N_{\text{total}}} \tag{4.26}$$

When the accuracy for multiple sequences shall be calculated, the accuracies of the individual sequences are weighted by their length and the average of those weighted values if computed. The combined accuracy for N_{seq} sequences with the length of sequence i being denoted as l_i can therefore be calculated according to the following equation.

$$a_{\text{comb}} = \frac{\sum_{i=1}^{N_{\text{seq}}} a_i \cdot l_i}{\sum_{i=1}^{N_{\text{seq}}} l_i} \tag{4.27}$$

Weighted F-Measure As the second performance measure, we use the *weighted F-measure* as defined in [6]. This measure improves over the accuracy by taking 'into account the precision and recall for each class and can [therefore] provide a better assessment of performance'[6, p. 2036]. The precision hereby describes the proportion of all classifications of a given class that were correct. Mathematically this is defined as follows, with N_{TP} denoting the number of correct classifications, or true positives of a given class. N_{FP} are the number of false classifications of the class, or false positives.

$$\textit{precision} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FP}}} \tag{4.28}$$

Recall on the other hand describes the proportion of the correctly predicted samples of that class, compared to all samples of said class. This is defined by the following equation, where N_{FN} describes the number of samples of the given class that were not classified correctly, also called the false negatives.

$$\textit{recall} = \frac{N_{\text{TP}}}{N_{\text{TP}} + N_{\text{FN}}} \tag{4.29}$$

Based on these two values for all N_{class} classes, the weighted F-measure F_1 can be computed according to the expression where w_i denotes the proportion of samples of class i in the data.

$$F_1 = \sum_{i=1}^{N_{\text{class}}} 2 \cdot w_i \cdot \frac{\textit{precision}_i \cdot \textit{recall}_i}{\textit{precision}_i + \textit{recall}_i} \tag{4.30}$$

If the weighted F-measure shall be calculated over multiple sequences, this can be done in the same way as already described for the accuracy. The weighted F-measure for each sequence individually is calculated and afterwards the weighted average of them is computed.

The accuracy and the weighted F-measure can now be calculated based on both, the segment and the sequence validation data. This gives us the four values a_{seg}, a_{seq}, $F_{1,\text{seg}}$ and $F_{1,\text{seq}}$ which we use for comparing the results of our experiments. The implementation of these described performance measures can be found in the appendix for this chapter.

4.3 Neural Networks for Activity Recognition

After having now laid out the foundations of feature extraction and ANNs, in this section we want to present how CNNs and LSTMs can be applied to the task of HAR. When looking at CNNs, we present a CNN architecture which is proposed in [13] and in the section after that we look at the LSTM architectures and training processes which we tested in our experiments.

4.3.1 CNNs for HAR

A popular approach for the task of HAR is the use of CNNs. This section gives a more detailed explanation on how CNNs are used for HAR on the example of the CNN2D architecture form [13]. This is also the approach that we mainly compare our results to, therefore we want to provide an understanding of how the CNNs are used on the data. As we did already mention when presenting the current state of research, [13] does propose an improved architecture over the CNN2D, which we are using as our point of comparison. Since rebuilding the CNN-pf and CNN-pff architectures

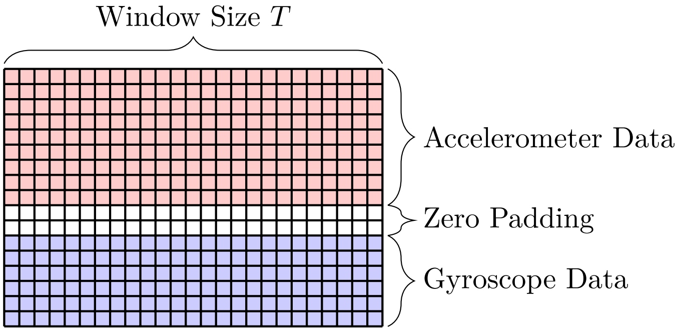


Figure 4.9: The grid arrangement of the input data for the CNN2D architecture.

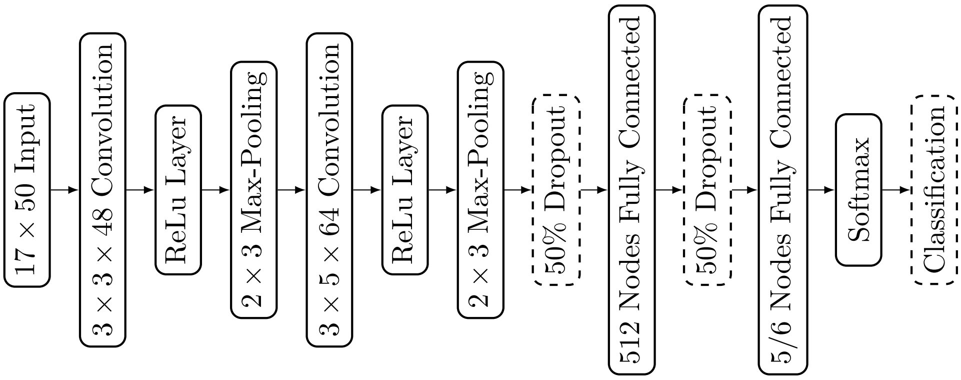


Figure 4.10: The layer structure of the CNN2D architecture.

would require modifying or writing our own implementation of the convolutional layer, we decided to only implement the more basic CNN2D architecture.

Since CNNs were originally designed to work with images as their input data, the time-series data of inertial sensors, which is the basis on which human activities shall be recognized, must be arranged into a grid before CNNs can be applied to it. For that purpose we do arrange our data into two-dimensional grids with a time axis along one of the sides of the grid and the different sensors arranged along the other side. The CNN2D architecture does only work with three acceleration and two gyroscope sensors. The approach furthermore implies to not convolute signals from sensors of different modalities, therefore a stripe of zeros is inserted between the signals of different modality. 'The number of zero-padded rows is set to one less than the vertical size of [the] 2D convolution kernel.'[13, p. 383] Since the two-dimensional convolution kernel of the first layer is of size 3 × 3, the final data grid is shown in Figure 4.9.
The grids need also to be of fixed width to be able to work on them with CNNs. For that, the data was segmented by a sliding window with length 60 samples and an overlap of 50 percent.

The CNN2D architecture consists of a grid input layer followed by the first of two convolutional layers. This layer uses 48 convolution kernels of size 3×3 and afterwards a ReLU activation function is applied. Before the data is fed into a second convolutional layer, maximum pooling with a pooling area of 2×3 is applies. The second convolutional layer uses 64 convolution kernels with a size of 3×5 and is also followed by a maximum pooling layer of the same size. Following that, a dropout with a 50 percent probability is introduced during training before feeding the data into a fully connected layer with 512 hidden units. After that layer, another dropout with 50 percent probability and another fully connected layer, with as many hidden units as there are activity classes, follow. As output layers, a softmax activation and a classification layer are applied. This layer structure is shown in Figure 4.10.

In contrast to the LSTM-based approaches which are outlined in the next section, the CNN2D or CNN1D architectures can not directly be applied to a complete sequence of data. The data needs to be segmented as explained above and therefore we only get a prediction of the network every 30 samples. To compare the two approaches in the next chapter, we reimplemented the CNN2D and CNN1D architectures in MATLAB. The implementation of the CNN2D can be found in the appendices, Section B.2. We evaluate the LSTMs on the full sequence and the CNNs on 60-sample-long windows in the way they would be applied on a smart wearable device.

4.3.2 LSTM Approach to HAR

In our Experiments we test the application of LSTMs for analysing inertial sensor data with the goal of activity recognition. The detailed architecture of the LSTMs and the training process is presented in this section.

As we already mentioned, all of our experiments were set up and executed in MATLAB. The company MathWorks does provide a Deep Learning Toolbox for MATLAB which contains implementations for different ANN layers. We built our networks out of these layers and trained them based on training algorithms which are also implemented in the toolbox.

Network Architecture

Our approach to HAR is centred around a LSTM network which is used to recognize the activities recorded in the data. In this section we present the architecture of said network.

The data is, as usual for a RNN, fed into the network sequentially. The first layer of the network is therefore a sequence input layer with as many input nodes as the input vector of the given dataset has values. After the input layer, a dropout with a probability of 50 percent is introduced for regularization during training. The main part of the network consists of the subsequent LSTM layers which we vary in their number of nodes and the number of layers. In the architecture study, which is later on described in greater detail, we varied the number of LSTM layers between two and five and did vary the sizes of each of those layers. The last layer before the output layers is a fully connected layer with five or six, depending on the number of classes in the dataset, nodes. A fully connected layer, as implemented in the MATLAB Toolbox, does not contain an activation function, but only the matrix multiplication. This can also be interpreted as a fully connected layer with the identity function as the activation function. In the architecture study, we did also replace the fully connected layer with another LSTM layer for some variants. To convert the hidden states of the last layer into a usable output, we do apply a softmax and a classification layer as the last two steps in the network.

We group the networks in our experiments into four basic architectures, which we named SequenceLSTM, SequenceLSTMnoFC, RelaxedLSTM and RelaxedLSTMnoFC. The SequenceLSTM is the most basic of the four architectures. Like all the other architectures as well, it is built up as shown in Figure 4.11, but the last layer before the softmax activation is always a fully connected layer with as many nodes as the dataset has activity classes. The SequenceLSTM is trained by a non-modified implementation of the cross entropy loss. A variation of the SequenceLSTM is the SequenceLSTMnoFC architecture. Differentiating the SequenceLSTMnoFC from the SequenceLSTM is that the fully connected layer is replaced with another LSTM layer that has, just like the fully connected layer of the SequenceLSTM, as many nodes as the dataset has classes. The two RelaxedLSTM variants differ from the SequenceLSTMs in their training procedure. The RelaxedLSTMs do not use the standard cross entropy loss, but calculate their loss based on the relaxed truth which is described later on. This is not necessarily an architecture difference, but since MATLAB implements the loss function in the classification layer and since the resulting networks are different, based on the different training procedures, we regard this as a separate architecture variant. The *noFC* suffix for the fourth architecture has the same meaning, of replacing the fully connected layer with a LSTM layer, as for the SequenceLSTM architectures.

The complete architecture is shown in Figure 4.11 and the implementation of the classes `AbstractHARNetwork` and `RelaxedLSTM`, representing one of the network variants, can be found in

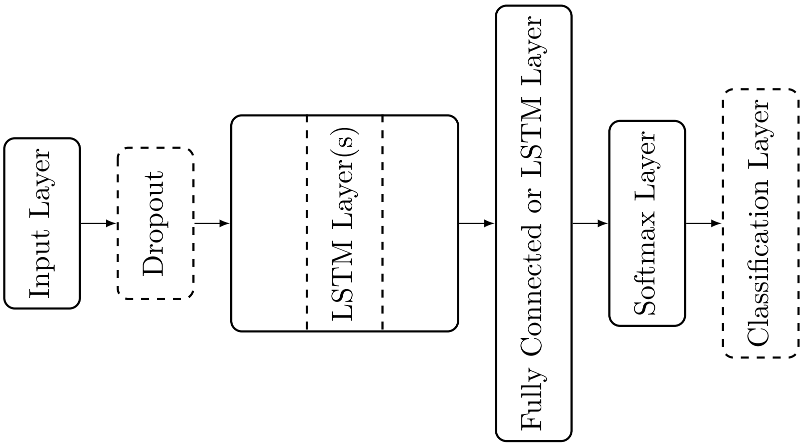


Figure 4.11: The layer structure of our approach to HAR.

the appendices, in Section B.2.

Training the LSTMs

Our first step of preparing the data for training, even before the feature extraction, was to divide the data into training, testing and validation subsets. We used 80 percent of the data for training and ten percent for each, the testing and validation data. The sequences of the datasets were randomly distributed into three sets with the given proportions, which left us with 19 training, three testing and two validation sequences for Opportunity. On mHealth, these proportions resulted in eight training, one testing and one validation sequence. The segmentation into these subsets was done before extracting the features for being able to normalize them only based on the training data.

After the feature extraction, we did then split the sequences at each change of a label to get sub-sequences which contain one activity class over their whole length. From these sub-sequences, windows with a fixed length of 200 samples and an overlap of 50 percent were cut. These were the windows which we used for our training data. From the steps described, we did obtain 3464 training sequences of length 200 samples for Opportunity and 9194 for mHealth respectively. The increased amount of training sequences on mHealth, even tough it contains fewer hours of data, stems from the higher sample rate that the data has been recorded at.

As the optimization algorithm for the training, we chose the *Adam* algorithm, which is an extension of stochastic gradient descent and was proposed in [17]. This algorithm has gained popularity for training ANNs over the last few years and is included in the MATLAB Deep Learning Toolbox. For that reason, it is easy to use for our experiments and promises good results.

We trained for a maximum of 100 epochs, but introduced an additional condition that would stop the training early if no progress is made, or we reached overfitting. For this, each epoch, the test data is passed trough the network and the loss as well as the accuracy for the test data is computed. From the loss and accuracy based on the test data (L_{test} and a_{test}) as well as the same values based on the training data (L_{train} and a_{train}), a performance value p for the network's current state is calculated, according to the following formula, after each epoch.

$$p = 3 \cdot \frac{3L_{\text{test}} + L_{\text{train}}}{4} - \frac{a_{\text{test}} + a_{\text{train}}}{2} \tag{4.31}$$

In this equation, the loss values are mainly influencing the performance value, with the loss based on the test data being weighted even more than the one based on the training data. The accuracy values in this equation range from zero to one, whilst the loss is typically in the range from zero to five. We check each epoch if the performance value has decreased and therefore improved and the training is stopped if the value does not improve for twelve consecutive epochs. The implementation of this early abort function can be found in the appendices of this work.

During training, we save a checkpoint of the network after each completed epoch. This allows us to select the best state of the network after the training has finished. We do again compute the performance values after each epoch according to 4.31 and do select the checkpoint with the best performance to be our result of that training pass.

Furthermore, we do train multiple instances of the same network from scratch and from those different results, we do also select the best network in the end. This is done because the learnable parameters of the network are being initialized at random. Therefore, it can happen that some training passes provide especially good or bad performance. How many training iterations can be performed for one network variant depends on the time and computational resources available.

We did also use L_2-regularization and a learning rate, which we drop to half of its value in regular intervals. This leaves us with four parameters which we have not specified yet. The learning rate α, the learning rate drop period T_α, the regularization parameter λ and the size of the mini batches N_{MB} for training with the Adam algorithm. To find the optimal values for these parameters, we conducted a training parameter study on the Opportunity dataset, which provided us with the values that we used for all the other experiments. This parameter study is described later on in Chapter 5.

Relaxed Truth Training

In our experiments, we did also test if we could optimize the training process of the LSTMs. For this purpose, we trained some out the networks in the later described architecture study with a modified training process. We present this modification to the training process, which we call *Relaxed Truth Training*, in this section.

As explained above, we train our networks based on sequences with a fixed length that contain only one activity class. This means that the label for the sequence is a constant value over all the 200 samples in the sequence. Since the softmax layer gives us the probability of all the possible classes as the output, this label needs to be converted into a probability of all the classes over time that can be used as the ground truth for the Cross Entropy Loss that is used in combination with the softmax layer. Therefore, the class label for each training sequence is converted into a probability sequence, as can be seen in the upper illustration in Figure 4.12, where the true class of the sequence gets a probability of one for the entire time of the sequence and the other classes all get a probability of zero associated with them. With that probability distribution, (4.12) can be applied to calculate the Cross Entropy loss based on the output $\hat{y}$ of the softmax layer and the ground truth y of the probability distribution.

The problem with this method is, that the network is forced to recognize the correct activity class instantly after one sample, because the probability of the true class is always one. Because the different classes can not be distinguished from only one sample of data, this is unrealistic. Coming from that point, we introduce a modification of the ground truth probabilities at the beginning of each sequence. We start with the idea, that at the beginning of the sequence, each activity class is equally likely to be the correct class. Therefore, we set the probability of all classes to $1/N_{\mathrm{class}}$, where N_{class} is the number of possible classes, for the first sample of the sequence. From that point on we do smoothly transition the probability distribution to the probability values which we want the network to actually produce for that sequence. This transition is done over a specified time T_{rel} and after that, the distribution of probability one for the true class and zero for all the others is kept. For this relaxation time, we did choose a quarter of our training sequence length and therefore 50 samples. To achieve a smooth transition of the probabilities and to keep the sum of all of them always at one, we used the shape of the sigmoid function for the changeover in the beginning of the sequence. The shape of the relaxed truth can be seen in the lower of the two plots in Figure 4.12.

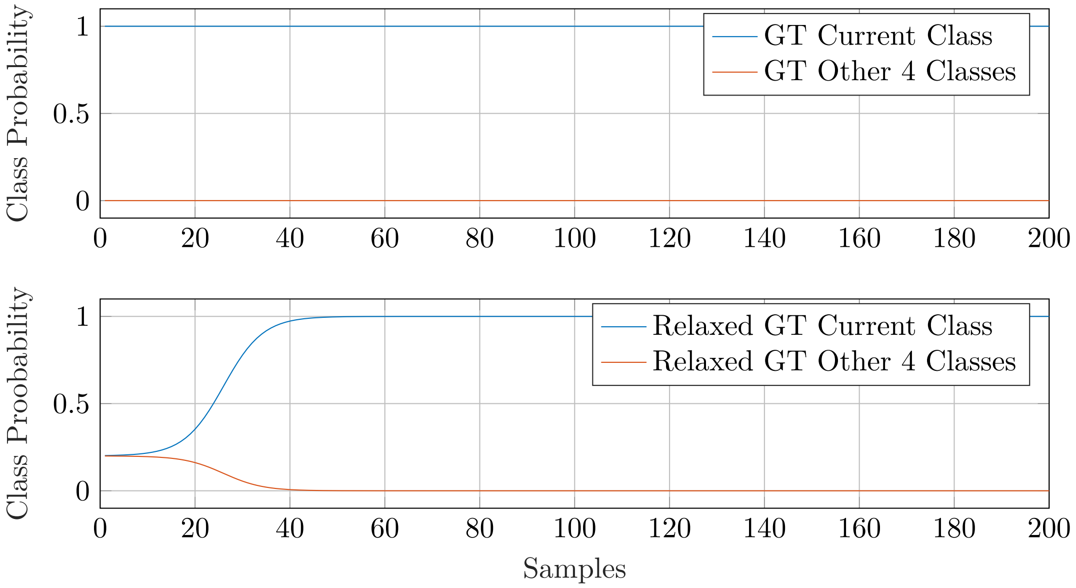


Figure 4.12: Strict ground truth (top) in comparison to the relaxed ground truth (bottom).

We implemented the relaxed truth training as a modification of the MATLAB classification layer. For this purpose, we did create a class which is derived from the `CrossEntropy` class out of the `nnet.internal.cnn.layer` package. In this class, we modified the methods `forwardLoss` and `backwardLoss` to use the relaxed instead of the strict truth. In addition to that, we did also need to modify the `ClassificationOutputLayer` class, to use our modified version of the `CrossEntropy` class as its internal layer. The relevant parts of these modifications and the implementation of the relaxed truth can be found in the appendices, Section B.2.

This chapter has now given an overview of all the methods that were used in the experiments to analyse the data. All information given here should suffice for understanding this work. If additional explanations are needed, they can be found in [1].

5 Experiments and Results

The purpose of this thesis is to compare LSTM variants to recent CNN solutions for HAR. We trained 190 LSTMs variants in two different ways and measured the result of the best LSTMs and CNNs on whole sequences. Our experiments were executed on the two different datasets Opportunity and mHealth, and we first present the results on Opportunity and afterwards validate them on mHealth. When doing that, we will compare our outcomes to results on the same data, that can be found in literature.

5.1 Experiments on Opportunity

The main part of our experiments was conducted on the Opportunity dataset. On it, we did the training parameter as well as the architecture study, which we will both describe in the following sections. All results, if not stated otherwise, are obtained from the same validation data set, which was not used before in any part of the training process. The two randomly selected sequences used as validation data are the fourth ADL and the drill run of subject two.

5.1.1 Learning Parameter Study

Before running any bigger experiments, we needed to determine the values for some training parameters that we used. The parameters in question were the following four: the learning rate α, the learning rate drop period T_α, the regularization parameter λ for L_2-regularization and the mini batch size N_{MB} to be used with the Adam algorithm.

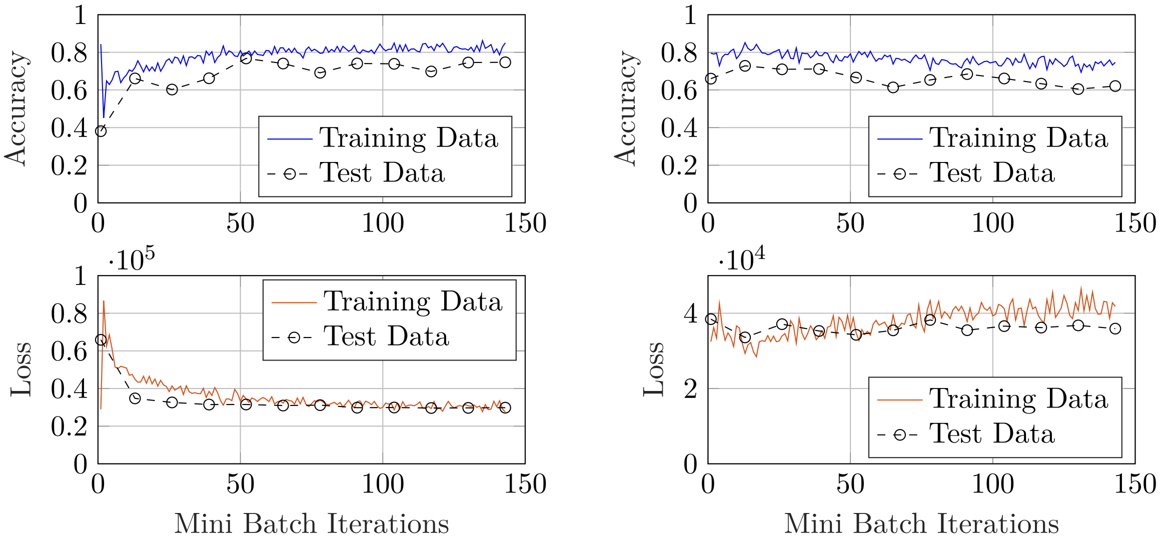
To limit the scope of this pre-study, we did select three values for each of the parameters that we were going to test, which gave us a total of $3^4 = 81$ parameter combinations to test. The values for the different parameters are listed in Table 5.1. All these combinations were tested for training two different networks. The first was a network with two LSTM layers of sizes 50 and 20, followed by a fully connected layer. The second network added a LSTM layer with 20 hidden nodes after the other two LSTM layers. All training processes were run once with relaxed truth training.

The findings from the learning parameter study were, that only a limited amount of parameter combinations did result in stable training progress. Only a single combination of learning rate drop period, regularization parameter and mini batch size resulted in the expected shape of the training accuracy rising in the shape of a limited growth and the loss decreasing and approaching zero. The learning rate seemed to not have a strong impact on the training performance, for all the three different learning rates yielded a usable training process if paired with the one specific combination of the other parameters.

The final parameters, which we decided to use for further experiments based on the training

Table 5.1: The values for the parameters learning rate α, learning rate drop period T_α, L_2-regularization λ and mini batch size N_{MB}, tested in the learning parameter study. The highlights mark the selected values.

Learning Parameter	α	T_α / Epochs	λ	N_{MB}
Values	0.05	*2*	0.005	*256*
	0.12	4	*0.0005*	512
	0.16	6	0.00005	1024



(a) The training performance with the selected training parameters.

(b) The training performance for $\alpha = 0.12$, $T_\alpha =$ 6 Epochs, $\lambda = 0.0005$, $N_{\mathrm{MB}} = 256$

Figure 5.1: Comparison of training progress of a RelaxedLSTM with two LSTM layers (50, 20) for the best and a bad combination of training parameters.

parameter study, were the following: learning rate α: 0.12, learning rate drop period T_α: 2 epochs, regularization parameter λ: 0.005, mini batch size N_{MB}: 256. These parameters resulted in the training progress shown in Figure 5.1a whilst Figure 5.1b shows the training progress for one of the combinations that did not yield stable training.

What we set out to achieve by testing the training parameters on two different networks was to find out if the parameters were usable, no matter the architecture of the network. We found, that the training performance of both networks was similar when using the same training parameters and therefore we assumed that the parameters selected would provide good training progress for the different architectures in the architecture study.

5.1.2 Architecture Study

After determining the learning parameters with the help of the learning parameter study, we did conduct an architecture study to compare the performance of different LSTM architectures. For this, we did test a total of 380 different architecture variants, based on 95 permutations of the four basic architectures. This section first outlines the scope of the architecture study and give an overview of the different network variants which we trained based on the data from the Opportunity dataset. Subsequently, we present and discuss the results of the experiment.

The basic architecture of the networks used in our experiments was already outlined in Section 4.3.2. While laying out this architecture, we did already mention the degrees of freedom which were varied during the architecture study. These are the number of LSTM layers and if the final layer, before the softmax activation function is applied, is a fully connected or another LSTM layer.

The 95 variants of our basic architectures vary in their number and size of the LSTM layers. We varied the number of LSTM layers between two and five and chose different numbers of hidden nodes for the layers. An overview of the different layer sizes used can be found in Table 5.2. The last column in that table lists the number of variations that result from all possible combinations of the selected sizes.

All 380 total networks were trained two times on the Opportunity dataset and the best network state out of these two training iterations was selected and saved. In the following, we want to present the results of these 380 networks, mainly by illustrating them in different plots.

Table 5.2: An overview of the layer variants trained for each of the basic architectures.

No. of LSTM-layers	Sizes for LSTM-layer No.					Number of variations
	1	2	3	4	5	
2	$\begin{bmatrix}100\\50\\30\end{bmatrix}$	$\begin{bmatrix}50\\30\\20\\10\end{bmatrix}$				12
3	$\begin{bmatrix}100\\50\\30\end{bmatrix}$	$\begin{bmatrix}50\\30\\20\end{bmatrix}$	$\begin{bmatrix}20\\10\\6\end{bmatrix}$			27
4	$\begin{bmatrix}100\\50\end{bmatrix}$	$\begin{bmatrix}50\\30\\20\end{bmatrix}$	$\begin{bmatrix}20\\10\end{bmatrix}$	$\begin{bmatrix}10\\6\end{bmatrix}$		24
5	$\begin{bmatrix}100\\50\end{bmatrix}$	$\begin{bmatrix}50\\30\end{bmatrix}$	$\begin{bmatrix}20\\10\end{bmatrix}$	$\begin{bmatrix}10\\6\end{bmatrix}$	$\begin{bmatrix}10\\6\end{bmatrix}$	32

To get a first impression of how the performance of all the trained networks is distributed, we can calculate the average μ and the standard deviation s of the performance for all 380 networks. When doing that for all four defined performance measures, we get the values that are shown in Table 5.3. In that table, it can be seen, that on average the networks perform better on the segmented data than on continuous sequences. This is to be expected, since we did train the LSTMs only on segmented data. It can also be seen that the calculated F-measures do differ from the accuracy values in a range of single digit percentages. This is also to be expected, since the weighted F-measure is only slightly different from the accuracy by taking precision and recall into consideration.

When looking at data segments containing only one activity class, the weighted F-measure is only composed of the F-score for the class of the current segment. This is because all the other classes are not present in the data segment and are therefore weighted with a factor of zero. Additionally, for the true class, the precision is always one, because there cannot be any false positives, since the segment does only contain the given class. Because of that, the F-measure on segmented data $F_{1,\text{seg}}$ does not provide any additional information above the accuracy based on that data a_{seg}, but only remap the accuracy according to $2 \cdot {}^{a_{\text{seg}}}\!/_{1+a_{\text{seg}}}$. For that reason, we do not look at this performance measure henceforth.

On data sequences that do not contain only one activity class, the F-measure does capture the balance between precision and recall, which is a better measure of performance for datasets with an unequal class distribution. When looking at the results in Table 5.3, it can be seen that for the sequence data the average F-measure is lower than the accuracy value on that data. This bears the information that, on average, the networks do not achieve an optimal balance between precision and recall.

Additionally, these averages and standard deviations can be calculated for smaller groupings of networks. Performance results for the network groups like the basic architectures or networks with the same number of hidden LSTM layers can be found in the appendix for this chapter.

The results for different groups of networks can also be illustrated as plots. Figure 5.2 shows the average performance, measured as the F-measure based on the sequence data, for the networks grouped by basic architecture and LSTM layer depth. In addition to that, the standard deviation

Table 5.3: The average and standard deviation performance for all networks in the architecture study.

Performance Measure ($\mu(s)$)							
$a_{\text{seg}}/\%$		$a_{\text{seq}}/\%$		$F_{1,\text{seg}}/\%$		$F_{1,\text{seq}}/\%$	
68.1	(7.3)	54.2	(7.8)	72.1	(7.9)	47.6	(10.6)

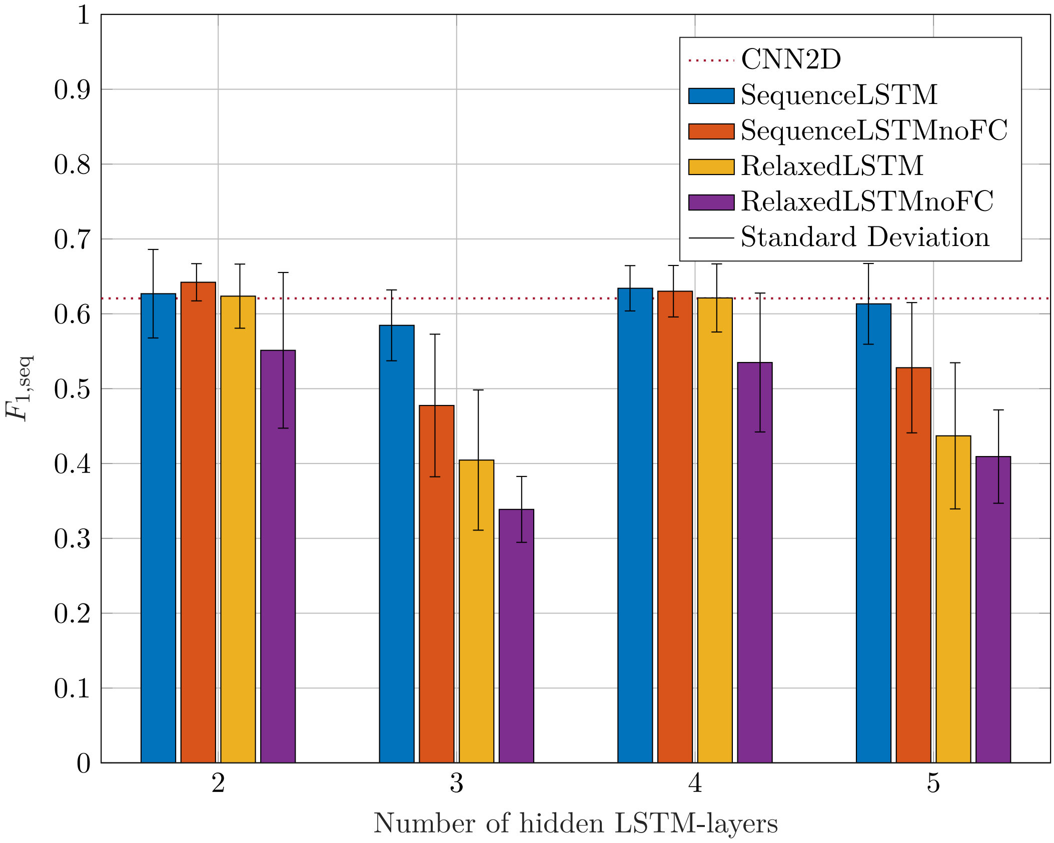


Figure 5.2: F-measure on sequence data for networks with different depth.

is marked at the top of the bars. Here it can be seen that an increase in network depth does not result in an accompanying increase in average performance. For the networks trained with the relaxed truth training, the opposite seems to even be the case. The average performance for them is decreasing with additional hidden LSTM layers. What however can be seen to roughly increase with the network depth is the standard deviation of the performance. From that we can tell that the larger a network gets, we have less ability to precisely predict how well the network will perform after training. This could either stem from different architectures with the same number of LSTM layers being suited better or worse for the task, or from a less stable training procedure for networks with higher depth.

To get an idea of how each individual one of the 380 networks does perform, we can plot the achieved weighted F-measure over the estimated complexity of the network, which is shown in Figure 5.3. The network complexity is hereby calculated as defined in Section 4.2.5. Here it can be seen that the highest performance achieved is around 70% (70.42%) and that there are many networks (54 different network) reaching an F-score which is higher than 65%. What cannot be seen is a trend of the performance increasing or decreasing with the complexity. From that, we can deduce that the size of the network, which is related to the networks computational complexity, is, in the size range which we tested in our experiments, not related to the classification performance that can be

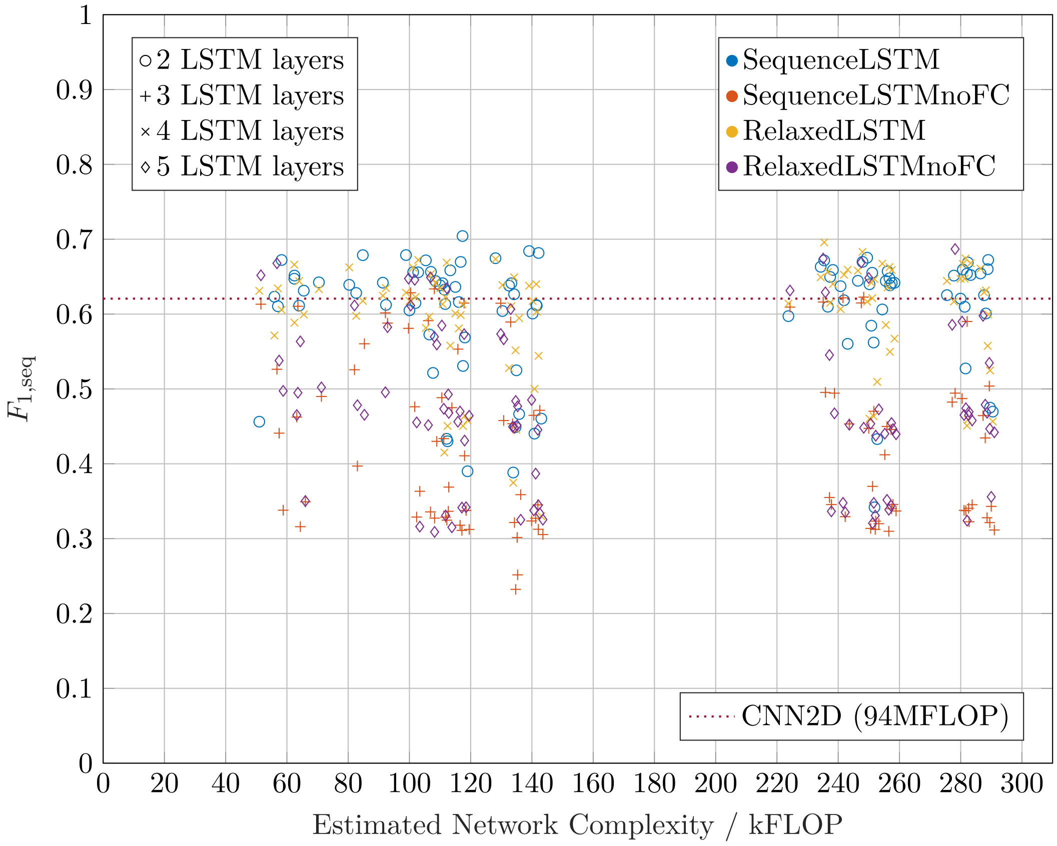


Figure 5.3: F-measure on sequence data over network complexity for different architectures.

achieved with the given network.

In addition to the complexity and the performance value of the networks, Figure 5.3 does also show the networks basic architecture and the number of hidden LSTM layers. This is done with the help of the shape and the colour of the scatter marks and two legends explain what basic architectures the colours and what number of hidden LSTM layers the shapes represent. From this additional information shown in the plot, it can be seen that different complexities can be achieved independently of the number of LSTM layers. This is done by varying the size of the layers, rather than their number. What can also be seen is that networks with fewer layers tend to perform better. This observation has already been made based on the information shown in Figure 5.2.

In the two plots which we last talked about, in addition to the performance of the trained LSTMs, a reference value is shown as a dotted line. This reference value is the performance of the CNN2D architecture from [13], which we laid out in the last chapter. In Figure 5.3 we can see that many of the trained LSTM networks do in fact provide a better performance value than the CNN. Based on the knowledge that the CNN has a computational complexity of 94MFLOP per classification and therefore is about two magnitudes more complex than the LSTMs, this can be seen as a success. Even when considering that the LSTMs are executed 30 times more often, they still require significantly less computational power. Additional performance values of the CNN2D and the CNN1D architectures from [13] are shown when comparing the LSTM performance to the CNNs.

More plots similar to the ones shown in Figures 5.2 and 5.3, based on the other performance measures, can be found in the appendix for this chapter.

When looking at Figure 5.2, one would think that the Relaxed Truth Training which we proposed does yield only worse or, at best, equal results to the non modified training equivalent. To actually evaluate the relaxed training process however, we need to compare the training course of two

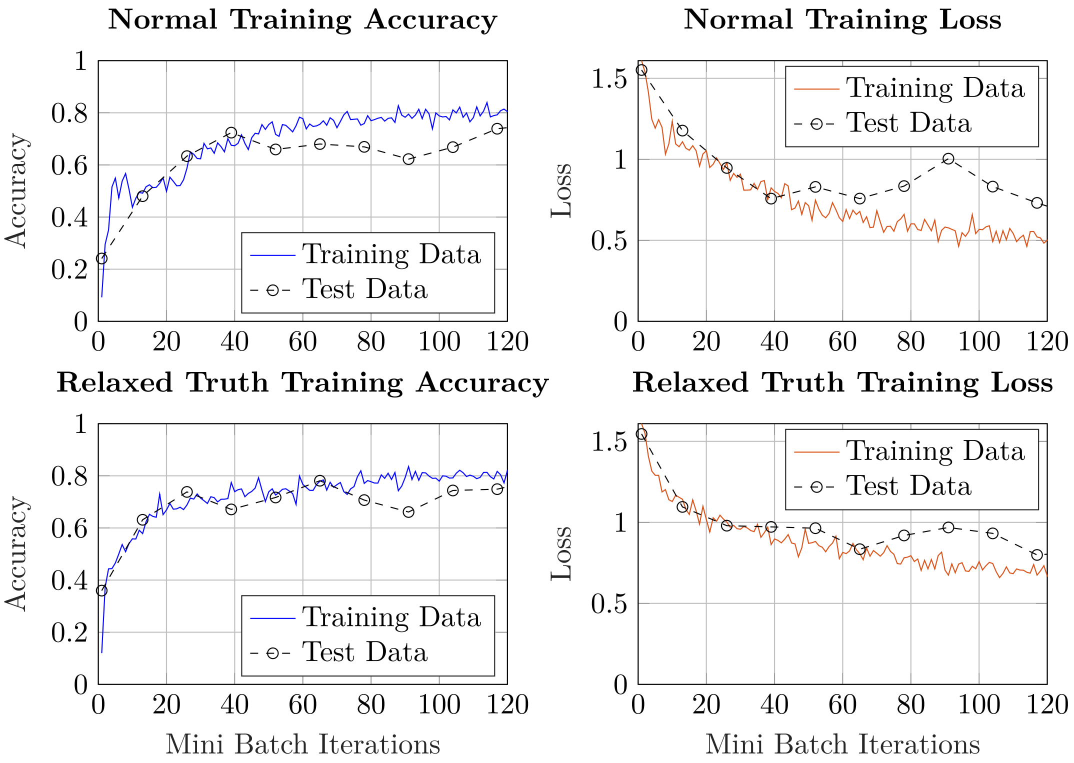


Figure 5.4: Comparison of the training progress between normal and relaxed truth training.

otherwise identical networks. For this, we chose a SequenceLSTM and a RelaxedLSTM with three LSTM layers with the sizes 50, 20 and 6 each. When comparing the performance for these two networks, we found that the results for all the performance measures were within two percent of one another. So the finished networks with normal and with relaxed truth training did perform almost identically. The advantage that can be gained however can be seen when we look at Figure 5.4, where the course of the accuracy and the loss over the first 120 training iterations is shown. In these plots we can compare how fast the accuracy increases and the loss decreases for the two different training methods. It can be seen that at the end of the 120 iterations, both networks perform with an accuracy of roughly 80 percent on the training data and the loss of the normally trained network is slightly lower with 0.5 compared to 0.7. The main difference when looking at the two plots of the accuracy is that the accuracy rises faster for that network that is being trained with relaxed truth training. This does only make a difference for the first iterations of training, but if only limited computational resources are available for training a network, one might be able to speed up the training process by using relaxed truth training.

The slightly higher loss values for the network that is trained with relaxed truth training can be explained by the fact that the calculation of the loss is not the same for the two networks. The network with normal training uses a Cross Entropy loss calculated based on the strict truth, whilst the other network uses the relaxed truth. When the network with Relaxed Truth Training does classify the current class confidently whilst still in the beginning of the segment were the truth is relaxed, it can increase the loss because it recognizes the class with too much confidence. The predicted class probabilities for a segment where this happened can be seen in Figure 5.5. Here the upper sub-plot shows the predicted probabilities for the five different classes whilst the lower sub-plot shows the target probabilities. It can be seen that until approximately sample 30, the prediction gives the current class a higher probability than the target probability. This increases the loss but does not decrease the accuracy. Therefore, it is the reason why we see a higher loss for the network that is trained with relaxed truth training in Figure 5.4.

For further investigation of the LSTMs trained in our architecture study and a detailed comparison

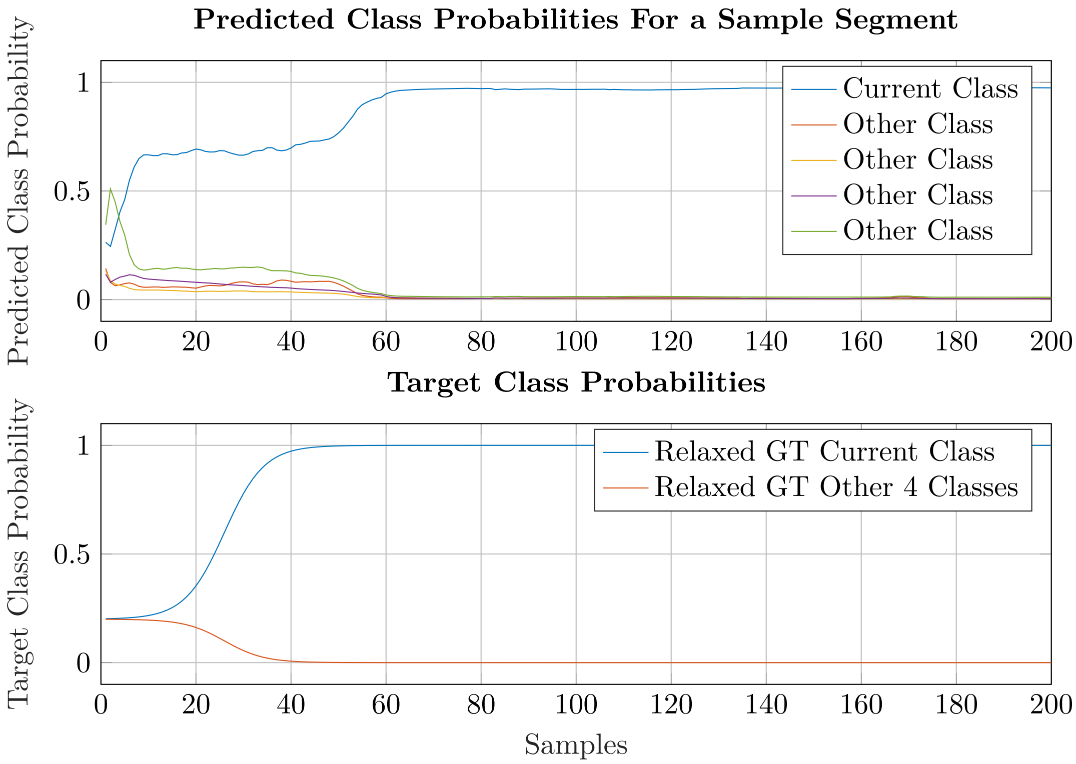


Figure 5.5: Comparison of predicted class probabilities and relaxed truth target. The higher predicted probability in the beginning leads to a higher loss.

to two CNN-based networks, we chose the two networks which did perform best on sequence or segmented data. The first network we chose was the network which achieved the highest value for the accuracy, as well as the F-measure based on sequence data. This network was a SequenceLSTM with four hidden LSTM layers where the LSTM layers were of the sizes 50, 30, 20 and 10. The resulting estimated complexity for this network are 117kFLOP per classification. As a second network, we selected the architecture that did perform the best based on segmented data. With a complexity of 248kFLOP per classification, this network was bigger than the first, even tough it did only contain two hidden LSTM layers. For that network, the final fully connected layer was also exchanged with a LSTM layer and it was trained with relaxed truth training. The sizes of the two main LSTM layers were 100 and 30 nodes.

In the next section we compare these two networks to CNNs based on continuous classification performance and other parameters.

5.1.3 Comparison to CNN Performance

In the following we compare the best two networks trained in the architecture study to two CNN architectures in three different ways. First we look at the performance values calculated from the classification results produced by the different networks. After that we shortly compare the computational complexity of the different networks and lastly we compare their performance in a continuous classification application which is close to real-life.

The four networks which we are comparing are a SequenceLSTM with four hidden LSTM layers, a RelaxedLSTMnoFC with two hidden LSTM layers, a CNN2D and a CNN1D network as defined in [13]. The selected LSTMs and how they were chosen was outlined in the last section and the CNNs were presented in Chapter 4 with a focus on the CNN2D architecture. More information on the two CNNs should be taken from [13]. In the following, the SequenceLSTM with LSTM layers of sizes 50, 30, 20 and 10 is called BestLSTMSeq to show that it was the LSTM performing best on sequence data. Respectively, the RelaxedLSTMnoFC with layer sizes 100 and 30 is called

Table 5.4: Comparison of two LSTMs and two CNNs on the Opportunity dataset.

NETWORK	PERFORMANCE		COMPUTATIONAL COMPLEXITY / kFLOP
	$a_{\text{seg}}/\%$	$F_{1,\text{seq}}/\%$	
BestLSTMSeq	75.2	70.4	117
BestLSTMSeg	78.8	66.9	248
CNN2D	75.6	62.1	94544
CNN1D	79.0	65.3	73634

BestLSTMSeg.

Table 5.4 shows an overview of the four networks and their performance. In this table we chose the accuracy on segmented data and the weighted F-measure on sequence data as the two performance measures because they do best capture the performance in those two scenarios. Here it can be seen that the best performance on segmented data was achieved by the CNN1D. Close behind that is the LSTM which did perform best on segmented data. When looking at the performance on continuous sequences of data, the BestLSTMSeq network outperforms the others by at least 5 percent points. This shows the advantage that LSTMs have over CNNs, which is the native applicability to continuous sequences. The CNNs can only be applied to data segments and might outperform the LSTMs when working on that form of data, but since the data of continuous sequences needs to be segmented for the CNNs to be able to work on it, all the context information outside the segment gets lost. This is why the BestLSTMSeq network is able to outperform the other networks in the sequence application.

In addition to that, the LSTMs provide an enormous advantage over the CNNs, which can be seen when looking at the last column in Table 5.4. Both LSTM networks are more than two orders of magnitude less computationally complex than even the smaller of the two CNNs. These numbers do not consider that the CNNs are run 30 times less often, but even when this is considered, the LSTMs are less computationally complex. Especially for applications on embedded devices with limited computational resources, this allows them to occupy less of the processor and therefore leave more capacity to run other tasks or use a more energy efficient processing unit.

In a real-life application, what a user sees of the classification of the model is not a condensed accuracy, but the course of the classification over time, compared to what he is actually doing. This can be simulated by running a continuous data sequence through the networks and looking at their classification output in comparison to the corresponding labels, which we call the ground truth. Such a classification of a whole sequence can be seen in the appendices for all four networks which we are looking at. This classification experiment was done on the fourth ADL run of subject two from the Opportunity dataset, which is an 18-minute-long sequence. In the following we want to look at two sections of the classification based on this sequence, which reach from seconds 150 to 300 and from 880 to 930. The classifications of the networks and the ground truth for these time segments can be seen in Figure 5.6. In each of the sub-plots, the blue line shows the label sequence of the dataset, which we regard as what is actually happening at any given time-stamp. Overlaid in orange is the activity classification of the networks or the predictions. The titles above the sub-plots show which network the prediction was produced by.

In the first time range of the sequence, it can be seen that the Null Class is labelled from seconds 165 to 285. This tells us that what the subject did in these two minutes cannot be classified as one of the labels walking, laying, sitting or standing. Being able to classify such negative samples, as well as instances that can be classified is important for a classification model to be useful. In the first sub-plot we see the classification results of the LSTM with the best results based on sequence

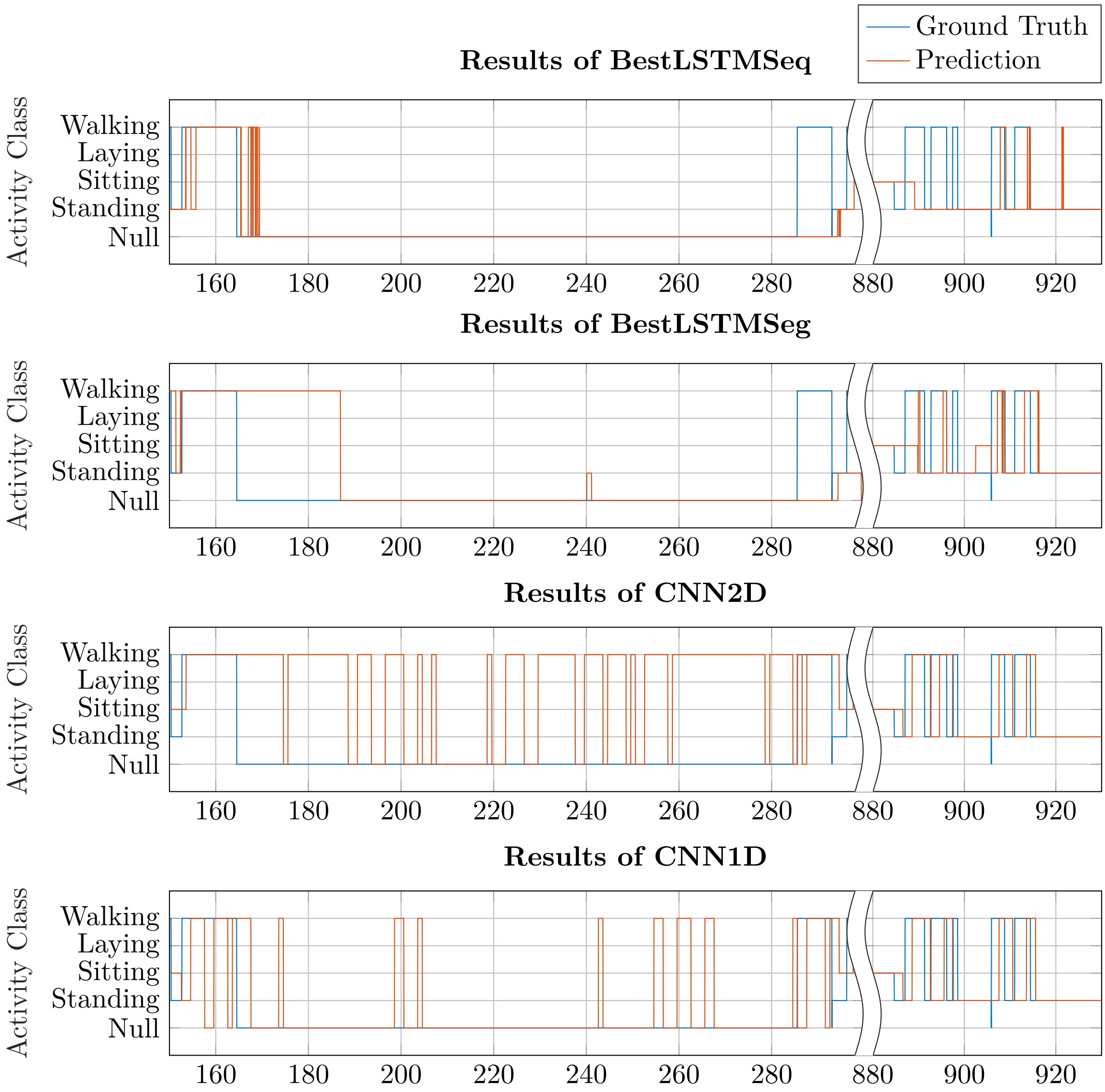


Figure 5.6: Continuous classification excerpt of two LSTMs and two CNNs on the Opportunity dataset.

data. It does recognize the Null Class in the given section reliably. Only within the first five seconds the output does jump to the walking class a few times. At the end of the Null Class segment, the output does change to other classes again, however the first section of walking is not detected. The other LSTM does perform equally well after the first third of the Null Class. From second 190 on the Null Class is recognized reliably except for a brief misclassification as standing. The first 20 seconds of Null Class however are misclassified as walking. When looking at the results of the CNNs which are shown in the lower two sub-plots, it can be seen that they are classifying the Null Class much more unreliably. Both networks show multiple seconds of misclassification as walking with the CNN2D correctly recognizing the Null Class less than half of the time. It is also notable that the classification of the CNNs appears to be much less stable.

In the second time range, the subject was mainly standing, which is interrupted by five short periods of walking. For these activities, the CNNs can be seen to perform better. Both of them are able to recognize four of the five walking periods and classify them with only a slight delay of about one second. The LSTMs do not perform as good as the CNN-based networks in this segment, with the BestLSTMSeg recognizing more of the walking, but mainly missing the first three instances and the BestLSTMSeq also mainly missing the last two instances.

From the observations made, we can now interpret what the main factors for these results were. The more stable classification of the LSTMs can be attributed to the ability to take context information of what happened before the current time stamp into account when classifying the activities. This allows the RNN-based networks to learn that activities are typically not changing rapidly but much more likely only every few seconds, and produce predictions based on this learning outcome. The CNNs do only have a section of 60 time-stamps to base their classification on. This leads to better performance on segmented data but, as we can see in Figure 5.6, less stable classifications in a close to real-life application. The fact that the Null Class is typically misclassified as walking by four independent networks indicates that what the subject is doing between seconds 165 and 185 seems to be similar to walking. When referring to the overview of the given sequence which can be found in the appendices, it can also be seen that the acceleration signals in this section show a highly dynamic signal that is similar to the one seen when the subject is walking. From the second time range it can be deduced that the LSTMs have trouble recognizing shorter periods of one activity. In that use case, the CNN-based networks are clearly superior with the only disadvantage shown being the delay in the classification, which arises from the fact that a significant amount of the activity class in question does first need to be present in the 80-sample-long classification window.

After this comparison of LSTM and CNN classification performance on Opportunity, in the next section we compare our results to the approaches presented in the context of the Opportunity challenge.

5.1.4 Comparison to Other Approaches

In this section we compare the results of our LSTM-based approach to activity recognition with other approaches that were tested on the same data. For comparison, we be use the results of the Opportunity challenge which were published in [6]. Out of the tasks that were part of the Opportunity challenge, we are comparing our approach to the results achieved for Task A: ‘multimodal activity recognition: modes of locomotion’, since this is what we were doing in our experiments. From the different results given in [6, Table 3], we chose to compare against the F-measure values including the Null Class, based on data from subject two. This is roughly equivalent to our F-measure based on sequence data $F_{1,\text{seq}}$, which is only based on slightly different sequences of the same subject.

The results of our networks, measured as the F-measure on sequence data, and the results of a selection of the approaches presented in [6], measured as the F-measure for Task A based on data of subject two, are put together in Table 5.5. The first group of methods hereby denote our approaches, the second group are approaches of the organizers of the Opportunity challenge and the third group are entries to the challenge. A table including all the approaches to the challenge can be found in

Table 5.5: Performance comparison with entries to the Opportunity challenge.

NETWORK/METHOD	PERFORMANCE
	$F_{1,\mathrm{seq}}/\%$
BestLSTMSeq	70
BestLSTMSeg	67
LDA	64
NCC	58
3 NN	86
SStar	61
MI	85
MU	57

the appendices.

From the table, it can be seen that our two LSTM networks were only outperformed by two of the selected approaches (3 NN and MI) from [6]. Even when taking all the approaches into account, only three of them score higher than our approaches. All approaches that outperformed our LSTMs were classifiers based on a k-Nearest Neighbours approach. It is notable that this is a relatively simple classification technique which seems to be sufficient for the given application.

The other approaches shown in Table 5.5 are based on Linear Discriminant Analysis (LDA), Nearest Centroid Classification (NCC), Support Vector Machines (SStar) and Decision Trees (MU).

Overall, outperforming most of the approaches that were part of the Opportunity challenge can be seen as a success.

After having shown our experiment process based on the Opportunity dataset, in the next section we present the comparison of LSTMs and CNNs again, but for networks which were trained and tested based on data from the mHealth dataset.

5.2 Validation on mHealth

After we had completed all the experiments on the Opportunity dataset, which we described in the previous sections, we took the two best LSTM networks, adjusted their architecture slightly and retrained them on the mHealth dataset. This section gives an overview of the results which we achieved with our approach on the mHealth dataset.

The motivation behind conducting additional experiments on another dataset was to see if the approaches which yielded good results on one dataset did also provide good performance on another data basis. For that reason we took the two networks which were found to perform best, either on segmented or sequence data and adjusted them to work on the mHealth dataset. All necessary adjustments were contained to the input and output layers, with the output layers gaining an additional node to accommodate for the additional activity class. The input layer was reduced in size to fit the features of fewer sensors in the mHealth dataset. Additional information about the activity classes and the sensors in the mHealth and the Opportunity dataset was already presented in Section 2.2.

The performance measures and the complexity of the two LSTM networks, as well as the same CNN architectures which we were already using as a reference, can be seen in Table 5.6. The reduction in network complexity for the LSTMs when comparing them to the same networks on the Opportunity dataset is mainly caused by the reduction of input features due to the reduced number of sensors in

Table 5.6: Comparison of two LSTMs and two CNNs on the mHealth dataset.

NETWORK	PERFORMANCE		COMPUTATIONAL COMPLEXITY / kFLOP
	$a_{\text{seg}}/\%$	$F_{1,\text{seq}}/\%$	
BestLSTMSeq	84.5	75.5	70
BestLSTMSeg	77.8	70.5	153
CNN2D	76.5	66.3	94674
CNN1D	74.0	66.9	73635

the mHealth dataset. When looking at the CNNs, one can see that they did only slightly change in complexity, which is only due to the additional output node for the mHealth dataset.

Regarding the performance of the networks trained and tested on the mHealth data, it can be seen that the BestLSTMSeq was outperforming all other networks, even the BestLSTMSeg on both types of data, segmented and sequence data. Both LSTMs achieved excellent results on sequence data but on segmented data the BestLSTMSeg did only slightly outperform both CNNs.

Based only on this data, the validation of the architectures on a different dataset was definitely successful, but the mHealth dataset bears a problem when training Machine Learning algorithms. This problem is that the dataset is acutely inhomogeneous, which means that the amounts of the activity classes are unbalanced. For our validation sequence from the mHealth dataset, the recording of subject nine, the Null Class is present almost three quarters of the entire time in the sequence. Therefore, even a classification of the Null Class for each time-stamp does already reach a sequence accuracy of over 74 percent and an F-measure of over 63 percent.

When looking at the continuous classification results displayed in Figure 5.7, one can see that this is in part what happened with the CNNs, mainly the CNN2D. This network does not show any classification change for nine of the twelve activity instances. The CNN1D does at least classify parts of five of the twelve instances whilst the BestLSTMSeq does manage to recognize half of the instances at least once. These results are not a good training outcome and should definitely be improved by training the networks on an equalized version of the dataset by removing some data points for the Null Class. Since we were only using the experiments on the mHealth dataset to validate if our networks which performed good on the Opportunity dataset did also achieve promising performance on other data, we did not invest the time to optimize the training process on the mHealth dataset. From the results, it can be seen that some activity classes are already recognized by our approach and this is enough to confidently say that good performance could be achieved on other datasets, if the time for optimizing the training process to the given dataset is invested.

In [13], classification results of over 95 percent accuracy are being reported for the CNN1D and CNN2D architectures on the data of subject nine of the mHealth dataset. These results do differ from ours because the networks in [13] were trained and tested only on the data of the true activity classes while leaving out the data from the Null Class. This eliminates the task of needing to distinguish the instances were an activity cannot be classified as one of the available labels and therefore does reduce the complexity of the task significantly.

These were all the experiments which we conducted to resolve the question if LSTMs are suited for analysing time-series data from inertial sensor units. In the next chapter we summarize and conclude all information presented in this work.

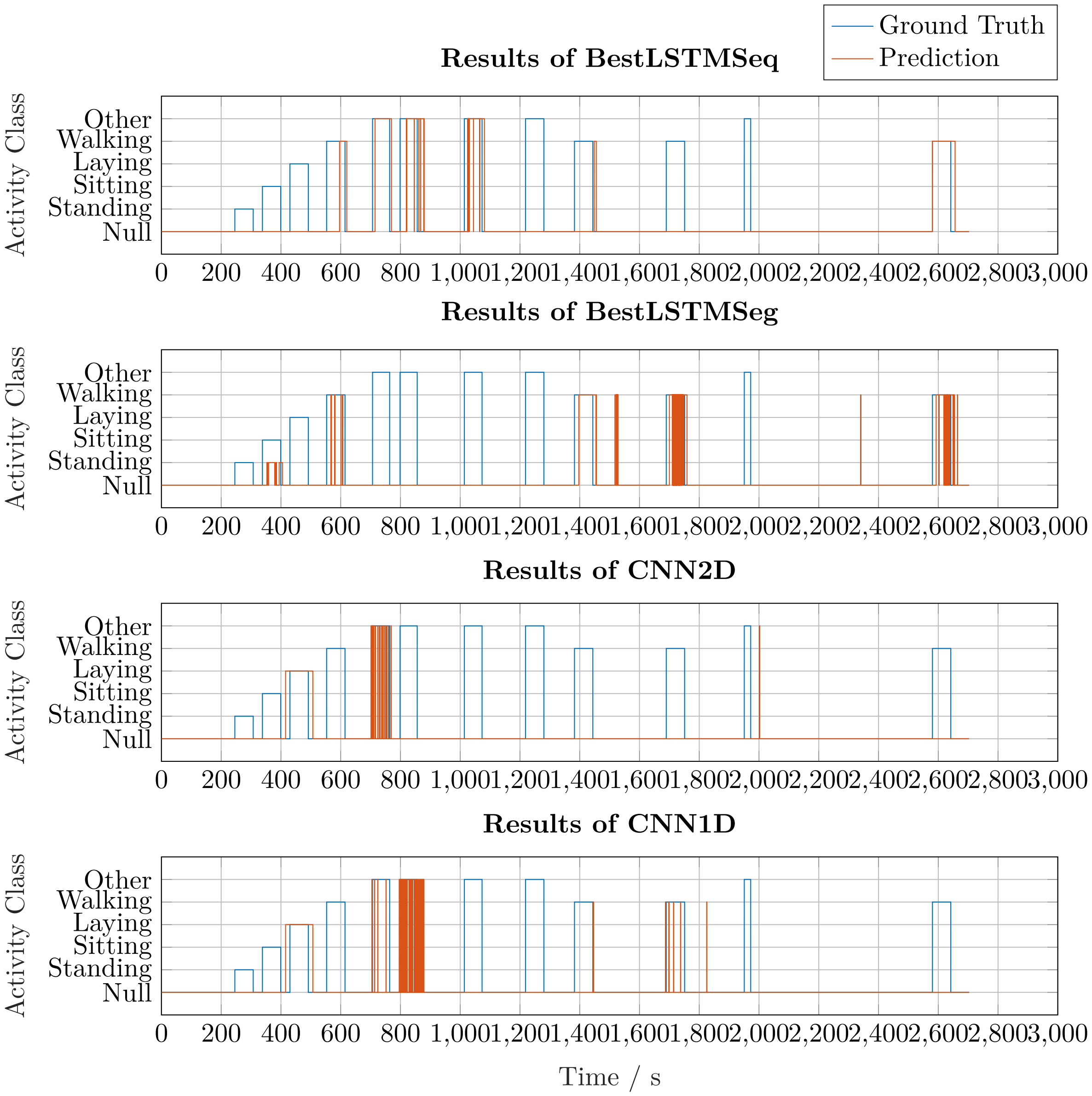


Figure 5.7: Continuous classification of two LSTMs and two CNNs on the mHealth dataset.

6 Conclusion

Under the research question ‘How do LSTMs perform compared to other approaches on time-series data from inertial sensors?’, this thesis set out to compare the performance of a LSTM-based approach to other Machine Learning techniques used for HAR. In presenting this work, first the field of HAR was defined and the datasets Opportunity and mHealth, as well as the current state of research was laid out. Connected to the datasets, we presented the sensors which make up an IMU and explained their functional principles. When laying out the methods of analysis, we focused on CNNs and RNNs, since the first of them are a common approach in current research but computationally expensive, whilst the latter were the focus of this work. After explaining feature preprocessing and the neural network architectures in general, we presented architectures based on both CNNs and LSTMs for the task of HAR. Finally, in the last chapter we presented our experiments, which mainly consisted of an architecture study where we compared 380 different LSTM variants to two CNN architectures. In addition to searching for the best suited LSTM architecture, we did also test a LSTM training optimization called Relaxed Truth Training.

In our experiments we found that LSTMs are indeed suited for the application of HAR with our two best LSTM-based networks outperforming the reference CNN architectures by over five percent points. Our best performing network was a SequenceLSTM with four LSTM layers of sizes 50, 30, 20 and 10, followed by a fully connected layer with as many nodes as the dataset has activity classes. Especially in our close to real-life task of continuous classification, this network showed exceptional performance for activities which continued for a longer period of time. When comparing our results to the entries to the Opportunity challenge presented in [6], we outperformed all but one entry and two approaches of the challenge organizers.

In addition to these results achieved on Opportunity, we were able to perform a basic validation on the mHealth dataset. We retrained our best-performing two architectures and saw promising results without investing much time into optimization of the training process of the new data basis.

Our training optimization Relaxed Truth Training was not able to improve the overall performance achieved by a LSTM network. However, it was able to show improved performance in the beginning of the training process. If only limited training time and computational resources are available for training a LSTM, this could therefore be a training optimization worth considering.

The motivation for this work was not to develop a new classification technique for human activities, but to validate the LSTM approach taken at Bosch for developing a theft detection algorithm for electric bikes. With our results in the field of HAR, we can now say that LSTMs can be generally used for analysing time series data of inertial sensors. The development of the theft detection algorithm can therefore confidently be continued.

Lastly we want to list some points where further research is needed but potential was shown to improve the performance of the networks in this work and also the theft detection algorithm.

As a first point, the hyperparameters of training can be optimized. We did only train on data segments with a fixed length and with only one activity class contained in them. When testing the networks on continuous sequences later on, we did therefore apply the network to class transients which they were not trained for. Training the networks for this situation can be done by training on sequences of varying length and sequences that do include class transitions.

Since more data does usually result in better performance for Machine Learning applications, this is another point where future work is possible. Since the two datasets used are not huge, recording a new dataset or combining existing ones to form a bigger one would provide a better foundation for training networks with more solid classification abilities.

Additional optimization potential can be found for the features. For reducing the network complexity, reducing the number of features would be a starting point that shows great potential since for now we did use as many features as possible. In addition to that, current research is focussing on automatic feature extraction, for example based on stacked autoencoders [7].

As we have seen both LSTMs and CNNs performing well for slightly different parts of the task of HAR, a combination of them can also be thought of. Initial research in the area of CNN-LSTMs has already been done [27] and this could also prove to be a useful approach to the analysis of time series sensor data.

Bibliography

[1] Charu C. Aggarwal. *Neural Networks and Deep Learning.* Springer International Publishing, 2018. DOI: `10.1007/978-3-319-94463-0`. URL: `https://doi.org/10.1007/978-3-319-94463-0`.

[2] Davide Anguita et al. 'A Public Domain Dataset for Human Activity Recognition using Smartphones'. In: Jan. 2013.

[3] Oresti Banos et al. 'Design, implementation and validation of a novel open framework for agile development of mobile health applications'. In: *BioMedical Engineering OnLine* 14.Suppl 2 (2015), S6. DOI: `10.1186/1475-925x-14-s2-s6`. URL: `https://doi.org/10.1186/1475-925x-14-s2-s6`.

[4] Oresti Banos et al. 'mHealthDroid: A Novel Framework for Agile Development of Mobile Health Applications'. In: *Ambient Assisted Living and Daily Activities.* Springer International Publishing, 2014, pp. 91–98. DOI: `10.1007/978-3-319-13105-4_14`. URL: `https://doi.org/10.1007/978-3-319-13105-4_14`.

[5] Adrian Burns et al. 'SHIMMER™ – A Wireless Sensor Platform for Noninvasive Biomedical Research'. In: *IEEE Sensors Journal* 10.9 (Sept. 2010), pp. 1527–1534. DOI: `10.1109/jsen.2010.2045498`. URL: `https://doi.org/10.1109/jsen.2010.2045498`.

[6] Ricardo Chavarriaga et al. 'The Opportunity challenge: A benchmark database for on-body sensor-based activity recognition'. In: *Pattern Recognition Letters* 34.15 (Nov. 2013), pp. 2033–2042. DOI: `10.1016/j.patrec.2012.12.014`. URL: `https://doi.org/10.1016/j.patrec.2012.12.014`.

[7] Belkacem Chikhaoui and Frank Gouineau. 'Towards Automatic Feature Extraction for Activity Recognition from Wearable Sensors: A Deep Learning Approach'. In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW).* IEEE, Nov. 2017. DOI: `10.1109/icdmw.2017.97`. URL: `https://doi.org/10.1109/icdmw.2017.97`.

[8] Wan-Young Chung, Amit Purwar and Annapurna Sharma. 'Frequency domain approach for activity classification using accelerometer'. In: *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society.* IEEE, Aug. 2008. DOI: `10.1109/iembs.2008.4649357`. URL: `https://doi.org/10.1109/iembs.2008.4649357`.

[9] A. Efimovskaya and A. M. Shkel. 'Multi-Degree-of-Freedom MEMS Coriolis Vibratory Gyroscopes Designed for Dynamic Range, Robustness, and Sensitivity'. In: *2018 DGON Inertial Sensors and Systems (ISS).* IEEE, Sept. 2018. DOI: `10.1109/inertialsensors.2018.8577146`. URL: `https://doi.org/10.1109/inertialsensors.2018.8577146`.

[10] Atis Elsts et al. 'On-Board Feature Extraction from Acceleration Data for Activity Recognition'. In: *EWSN '18.* Association for Computing Machinery (ACM), Feb. 2018, pp. 163–186. ISBN: 9780994988621.

[11] Shurui Fan, Yating Jia and Congyue Jia. 'A Feature Selection and Classification Method for Activity Recognition Based on an Inertial Sensing Unit'. In: *Information* 10.10 (Sept. 2019), p. 290. DOI: `10.3390/info10100290`. URL: `https://doi.org/10.3390/info10100290`.

[12] Thomas Frey and Martin Bossert. *Signal- und Systemtheorie.* Vieweg+Teubner, 2009. DOI: `10.1007/978-3-8348-9292-8`. URL: `https://doi.org/10.1007/978-3-8348-9292-8`.

[13] Sojeong Ha and Seungjin Choi. 'Convolutional neural networks for human activity recognition using multiple accelerometer and gyroscope sensors'. In: *2016 International Joint Conference on Neural Networks (IJCNN).* IEEE, July 2016. DOI: `10.1109/ijcnn.2016.7727224`. URL: `https://doi.org/10.1109/ijcnn.2016.7727224`.

[14] Ekbert Hering and Gert Schönfelder, eds. *Sensoren in Wissenschaft und Technik*. Vieweg+ Teubner Verlag, 2012. DOI: `10.1007/978-3-8348-8635-4`. URL: `https://doi.org/10.1007/978-3-8348-8635-4`.

[15] D. H. Hubel and T. N. Wiesel. 'Receptive fields of single neurones in the cat's striate cortex'. In: *The Journal of Physiology* 148.3 (Oct. 1959), pp. 574–591. DOI: `10.1113/jphysiol.1959.sp006308`. URL: `https://doi.org/10.1113/jphysiol.1959.sp006308`.

[16] Mridul Khan et al. 'A Feature Extraction Method for Realtime Human Activity Recognition on Cell Phones'. In: (Oct. 2019).

[17] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: `1412.6980 [cs.LG]`.

[18] Emiro De-La-Hoz-Franco et al. 'Sensor-Based Datasets for Human Activity Recognition – A Systematic Review of Literature'. In: *IEEE Access* 6 (2018), pp. 59192–59210. DOI: `10.1109/access.2018.2873502`. URL: `https://doi.org/10.1109/access.2018.2873502`.

[19] J. Mantyjarvi, J. Himberg and T. Seppanen. 'Recognizing human motion with multiple acceleration sensors'. In: *2001 IEEE International Conference on Systems, Man and Cybernetics. e-Systems and e-Man for Cybernetics in Cyberspace (Cat.No.01CH37236)*. Vol. 2. IEEE, Oct. 2001. DOI: `10.1109/icsmc.2001.973004`. URL: `https://doi.org/10.1109/icsmc.2001.973004`.

[20] M. J. Mathie et al. 'Detection of daily physical activities using a triaxial accelerometer'. In: *Medical & Biological Engineering & Computing* 41.3 (May 2003), pp. 296–301. DOI: `10.1007/bf02348434`. URL: `https://doi.org/10.1007/bf02348434`.

[21] Morgan T. Redfield et al. 'Classifying prosthetic use via accelerometry in persons with transtibial amputations'. In: *Journal of Rehabilitation Research and Development* 50.9 (2013), pp. 1201–1212. DOI: `10.1682/jrrd.2012.12.0233`. URL: `https://doi.org/10.1682/jrrd.2012.12.0233`.

[22] Paul P. L. [Verfasser/in] Regtien, ed. *Sensors for mechatronics*. 1. ed. Elsevier insights. Amsterdam ; Heidelberg [u.a.]: Elsevier, 2012, XII, 310 Seiten. ISBN: 978-0-1239-1497-2.

[23] Leopoldo Marchiori Rodrigues and Mario Mestria. 'Classification methods based on bayes and neural networks for human activity recognition'. In: *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*. IEEE, Aug. 2016. DOI: `10.1109/fskd.2016.7603339`. URL: `https://doi.org/10.1109/fskd.2016.7603339`.

[24] Daniel Roggen et al. 'An Educational and Research Kit for Activity and Context Recognition from On-body Sensors'. In: *2010 International Conference on Body Sensor Networks*. IEEE, June 2010. DOI: `10.1109/bsn.2010.35`. URL: `https://doi.org/10.1109/bsn.2010.35`.

[25] Daniel Roggen et al. 'Collecting complex activity datasets in highly rich networked sensor environments'. In: *2010 Seventh International Conference on Networked Sensing Systems (INSS)*. IEEE, June 2010. DOI: `10.1109/inss.2010.5573462`. URL: `https://doi.org/10.1109/inss.2010.5573462`.

[26] Charissa Ann Ronao and Sung-Bae Cho. 'Human activity recognition using smartphone sensors with two-stage continuous hidden Markov models'. In: *2014 10th International Conference on Natural Computation (ICNC)*. IEEE, Aug. 2014. DOI: `10.1109/icnc.2014.6975918`. URL: `https://doi.org/10.1109/icnc.2014.6975918`.

[27] Tara N. Sainath et al. 'Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks'. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, Apr. 2015. DOI: `10.1109/icassp.2015.7178838`. URL: `https://doi.org/10.1109/icassp.2015.7178838`.

[28] C. E. Shannon. 'A Mathematical Theory of Communication'. In: *Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. DOI: `10.1002/j.1538-7305.1948.tb01338.x`. URL: `https://doi.org/10.1002/j.1538-7305.1948.tb01338.x`.

[29] P. Stoica and R.L. Moses. *Spectral Analysis of Signals*. Pearson Prentice Hall, 2005. ISBN: 9780131139565. URL: `https://books.google.ca/books?id=h78ZAQAAIAAJ`.

[30] Wallace Ugulino et al. 'Wearable Computing: Accelerometers' Data Classification of Body Postures and Movements'. In: *Advances in Artificial Intelligence - SBIA 2012*. Springer Berlin Heidelberg, 2012, pp. 52–61. DOI: `10.1007/978-3-642-34459-6_6`. URL: `https://doi.org/10.1007/978-3-642-34459-6_6`.

[31] Michalis Vrigkas, Christophoros Nikou and Ioannis A. Kakadiaris. 'A Review of Human Activity Recognition Methods'. In: *Frontiers in Robotics and AI* 2 (Nov. 2015). DOI: `10.3389/frobt.2015.00028`. URL: `https://doi.org/10.3389/frobt.2015.00028`.

[32] Jhun-Ying Yang, Jeen-Shing Wang and Yen-Ping Chen. 'Using acceleration measurements for activity recognition: An effective learning algorithm for constructing neural classifiers'. In: *Pattern Recognition Letters* 29.16 (Dec. 2008), pp. 2213–2220. DOI: `10.1016/j.patrec.2008.08.002`. URL: `https://doi.org/10.1016/j.patrec.2008.08.002`.

[33] Ziyu Ye et al. *Comparison of Neural Network Architectures for Spectrum Sensing*. 2019. arXiv: `1907.07321 [eess.SP]`.

Appendices

List of Appended Figures

A.1 An overview of a complete ADL-run in the Opportunity dataset. XX
A.2 An overview of a complete sequence in the mHealth dataset. XXI

C.1 Accuracy on segmented data over network complexity for different architectures. XXXVI
C.2 F-measure on segmented data over network complexity for different architectures. XXXVIII
C.3 Accuracy on sequence data over network complexity for different architectures. XXXVIII
C.4 Accuracy on segmented data over network depth for different architectures. . XXXIX
C.5 F-measure on segmented data over network depth for different architectures. XXXIX
C.6 Accuracy on sequence data over network depth for different architectures. . . XL
C.7 Continuous classification on the Opportunity dataset. XLI

List of Appended Tables

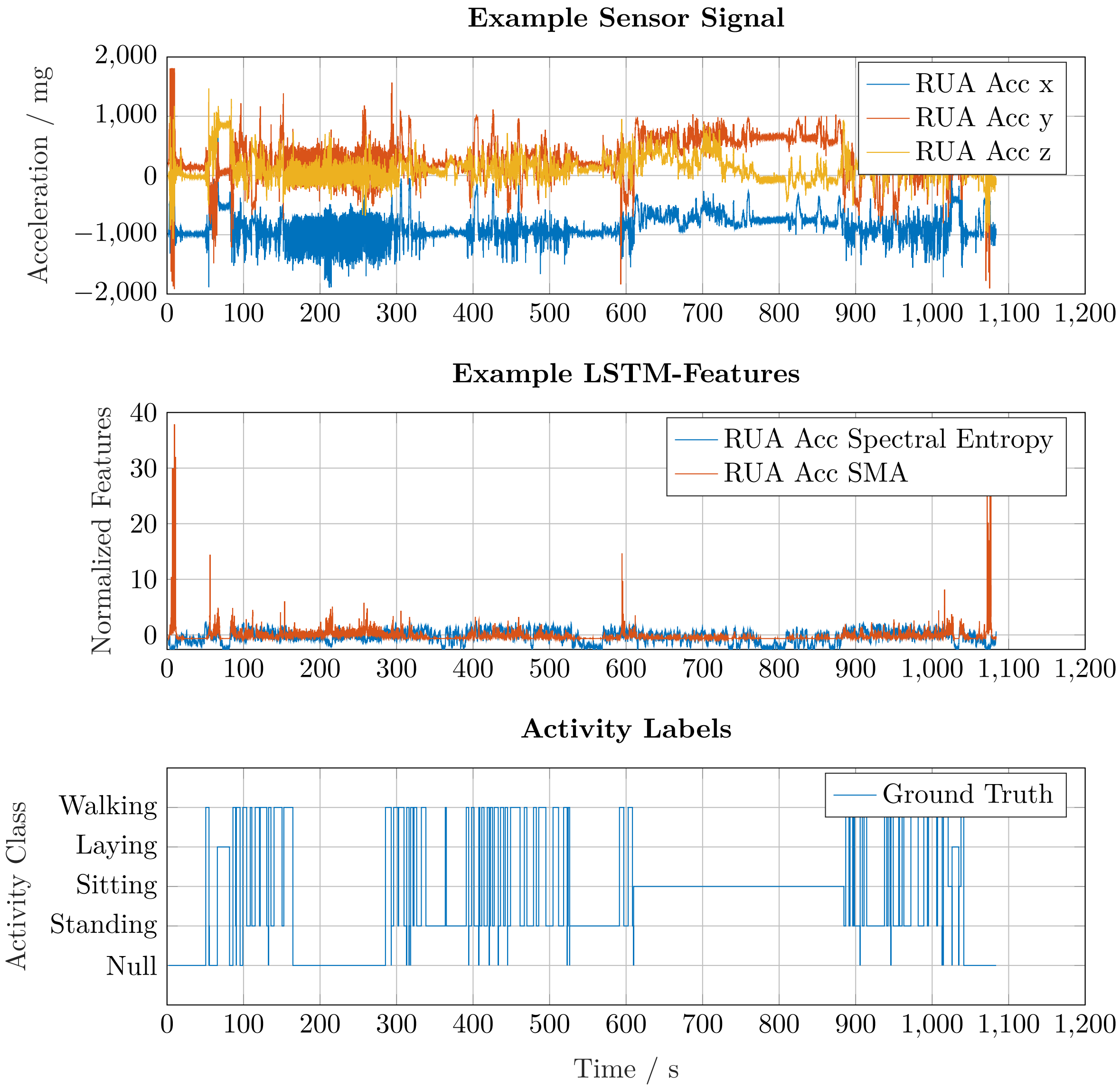
C.1 Performance results for different groups of networks in the architecture study. XXXVII
C.2 Performance comparison with entries to the Opportunity challenge. XLII

List of Code Listings

B.1 Implementation of the signal magnitude area. XXV
B.2 Calculation of the frequency power spectrum. XXV
B.3 Implementation of the spectral entropy. XXV
B.4 Modification to the loss calculation for relaxed truth training. XXV
B.5 Implementation of the conversion from strict to relaxed truth. XXVII
B.6 Implementation of the accuracy measure. XXVII
B.7 Implementation of the weighted and non-weighted F-measure. XXVII
B.8 MATLAB implementation of the CNN2D architecture. XXVIII
B.9 The base class for all our network implementations. XXIX
B.10 Implementation of one of our four basic LSTM architectures. XXXIII
B.11 Function for early training abortion. XXXIV

A Appendix for Chapter 2

This appendix holds additional information for the chapter on HAR. The content are additional plots illustrating the data that can be found in the datasets.



Example Sensor Signal

Example LSTM-Features

Activity Labels

Figure A.1: An overview of a complete ADL-run in the Opportunity dataset.

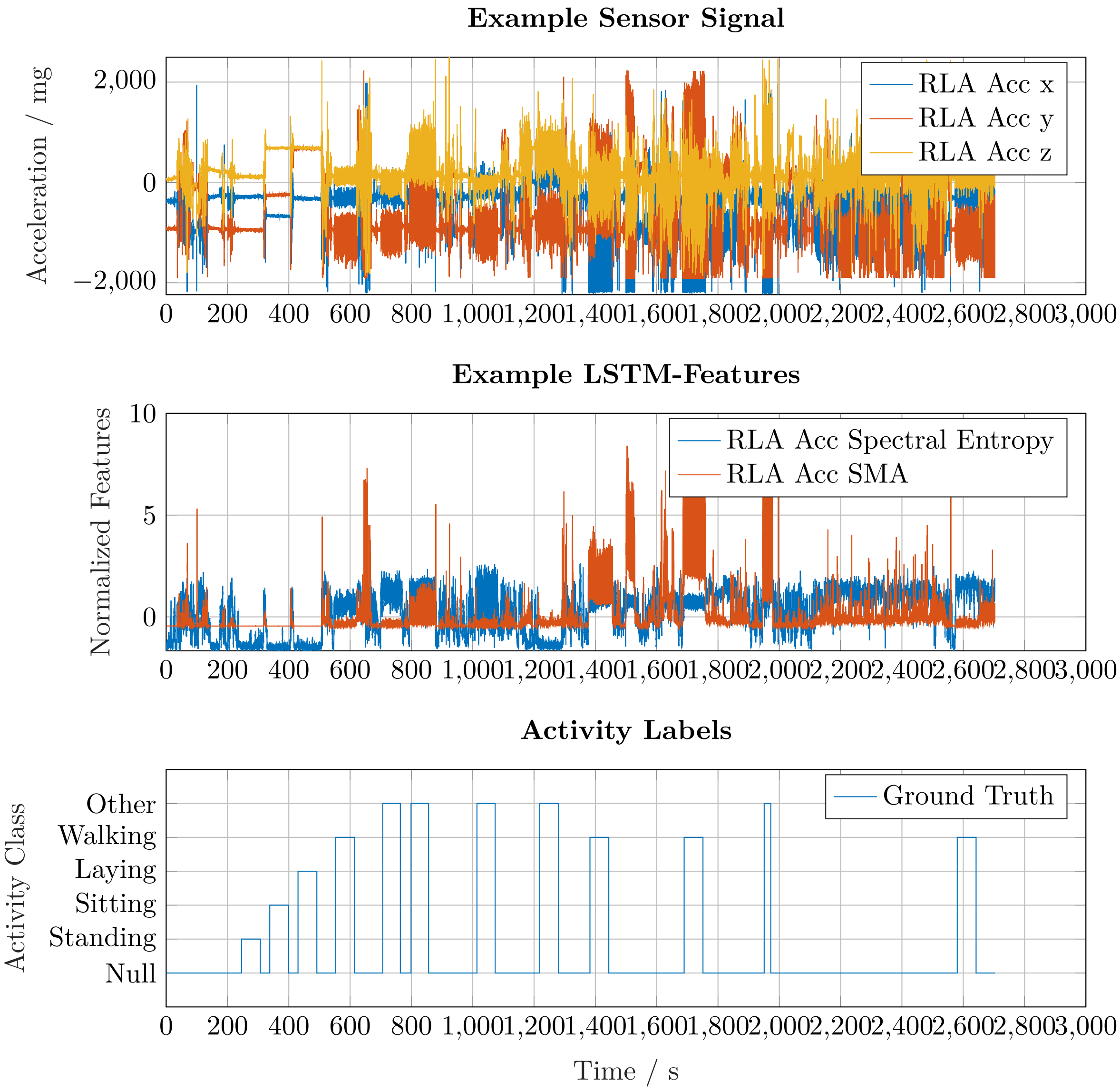


Figure A.2: An overview of a complete sequence in the mHealth dataset.

B Appendix for Chapter 4

This appendix contains additional information and code excerpts to illustrate the methods laid out in Chapter 4.

B.1 Comparison of the Backpropagation Through Time in RNNs and LSTMs

In the following, we want to compare the gradients in a vanilla RNN and a LSTM. For this, we look at a network with one single recurrent layer. This architecture can be seen in Figure 4.7b. The recurrent layer is then either a simple RNN, or a LSTM layer. For both, the RNN and the LSTM, the calculation of the output y_t from the hidden state h_t follows (4.17).

The gradients ar calculated from the loss, which can either be the Cross Entropy loss as expressed in (4.12) or the simpler squared loss, which is calculated by the following equation.

$$L_{\mathrm{sq}} = (y - \hat{y})^2 \tag{B.1}$$

B.1.1 RNN

In the following we want to look at the computation of the gradient in the vanilla RNN at time-stamp T. The computation of the recurrent part of the network is described as in (4.16). We introduce an expression for the combined weights as follows: $W = [W_{xh}, W_{hh}]$. This allows us to write the backpropagation through the network and through time as the following expression.

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial y_T}\frac{\partial y_T}{\partial h_T}\left(\prod_{t=2}^{T}\frac{\partial h_t}{\partial h_{t-1}}\right)\frac{\partial h_1}{\partial W} \tag{B.2}$$

As we will see later, the only difference for the LSTM is in the product expression in parentheses. Therefore, next we expand the expression in the product.

$$\begin{aligned}\frac{\partial h_t}{\partial h_{t-1}} &= \Phi'(W_{xh}x_t + W_{hh}h_{t-1}) \cdot \frac{\partial}{\partial h_{t-1}}(W_{xh}x_t + W_{hh}h_{t-1}) \\ &= \Phi'(W_{xh}x_t + W_{hh}h_{t-1}) \cdot W_{hh}\end{aligned} \tag{B.3}$$

If we now put (B.2) and (B.3) together, we get the following.

$$\begin{aligned}\frac{\partial L_T}{\partial W} &= \frac{\partial L_T}{\partial y_T}\frac{\partial y_T}{\partial h_T}\left(\prod_{t=2}^{T}\Phi'(W_{xh}x_t + W_{hh}h_{t-1}) \cdot W_{hh}\right)\frac{\partial h_1}{\partial W} \\ &= \frac{\partial L_T}{\partial y_T}\frac{\partial y_T}{\partial h_T}W_{hh}^{T-1}\left(\prod_{t=2}^{T}\Phi'(W_{xh}x_t + W_{hh}h_{t-1})\right)\frac{\partial h_1}{\partial W}\end{aligned} \tag{B.4}$$

This is the gradient of a vanilla RNN, which is later on compared with the gradient of a LSTM

B.1.2 LSTM

We want to now look at a network which is structured as the RNN, only the vanilla recurrent layer is replaced with a LSTM layer. The computation of the recurrent layer can therefore be expressed as the following six equations.

$$i_t = \Phi_g(W_{xh,i}x_t + W_{hh,i}h_{t-1}) \tag{B.5}$$
$$f_t = \Phi_g(W_{xh,f}x_t + W_{hh,f}h_{t-1}) \tag{B.6}$$
$$o_t = \Phi_g(W_{xh,o}x_t + W_{hh,o}h_{t-1}) \tag{B.7}$$
$$g_t = \Phi_c(W_{xh,g}x_t + W_{hh,g}h_{t-1}) \tag{B.8}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{B.9}$$
$$h_t = o_t \odot \Phi_c(c_t) \tag{B.10}$$

We do combine the weights again into one weight matrix W.

$$W = \begin{bmatrix} W_{xh,i} & W_{hh,i} \\ W_{xh,f} & W_{hh,f} \\ W_{xh,o} & W_{hh,o} \\ W_{xh,g} & W_{hh,g} \end{bmatrix}$$

Since the LSTM adds the additional cell state, we need to modify (B.2) and get the following expression for the backpropagation.

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial y_T}\frac{\partial y_T}{\partial h_T}\frac{\partial h_T}{\partial c_T}\left(\prod_{t=2}^{T}\frac{\partial c_t}{\partial c_{t-1}}\right)\frac{\partial c_1}{\partial W} \tag{B.11}$$

Here we see the additional expression $\frac{\partial h_T}{\partial c_T}$ which stems from (B.10). This can be expanded to the following.

$$\begin{aligned} \frac{\partial h_T}{\partial c_T} &= \frac{\partial}{\partial c_T} o_T \odot \Phi_c(c_T) \\ &= \frac{\partial o_T}{\partial c_T} \odot \Phi_c(c_T) + o_T \odot \Phi_c'(c_T) \end{aligned} \tag{B.12}$$

The partial derivative of the output gate regarding the cell state can be expanded in the following way.

$$\begin{aligned} \frac{\partial o_T}{\partial c_T} &= \frac{\partial}{\partial c_T}\Phi_g(W_{xh,o}x_T + W_{hh,o}h_{T-1}) \\ &= \Phi_g'(W_{xh,o}x_T + W_{hh,o}h_{T-1}) \odot \frac{\partial}{\partial c_T}(W_{xh,o}x_T + W_{hh,o}h_{T-1}) \\ &= 0 \end{aligned} \tag{B.13}$$

Where the second step is done because the inner part of the parentheses is not dependent on the cell state at time stamp T. If we now insert this result into (B.12) we get the following simplified expression.

$$\frac{\partial h_T}{\partial c_T} = o_T \odot \Phi_c'(c_T) \tag{B.14}$$

This expression can take on values between 0 and 1 for the output gate is calculated through the gate activation function Φ_g, which is typically the sigmoid function which can take on values between 0 and 1. The derivative of the cell activation function Φ_c is the derivative of the hyperbolic tangent function, which can also take on values between 0 and 1. This factor does therefore not provide a significant difference between the gradients of the RNN and the LSTM. The other difference is the expression in the multiplication from time-stamp 2 until the end. Here, we can do the following expansion.

$$\begin{aligned}\frac{\partial c_t}{\partial c_{t-1}} &= \frac{\partial}{\partial c_{t-1}}(f_t \odot c_{t-1} + i_t \odot g_t)\\ &= f_t \cdot \frac{\partial c_{t-1}}{\partial c_{t-1}} \odot \frac{\partial f_t}{\partial c_{t-1}} \cdot c_{t-1} + \frac{\partial i_t}{\partial c_{t-1}} \odot g_t + i_t \odot \frac{\partial g_t}{\partial c_{t-1}}\end{aligned} \tag{B.15}$$

Next we expand the derivative of the input gate.

$$\begin{aligned}\frac{\partial i_t}{\partial c_{t-1}} &= \frac{\partial}{\partial c_{t-1}}\Phi_g(W_{xh,i}x_t + W_{hh,i}h_{t-1})\\ &= \Phi_g'(W_{xh,i}x_t + W_{hh,i}h_{t-1}) \odot \frac{\partial}{\partial c_{t-1}}(W_{xh,i}x_t + W_{hh,i}h_{t-1})\\ &= \Phi_g'(W_{xh,i}x_t + W_{hh,i}h_{t-1}) \odot \left(\frac{\partial o_{t-1}}{\partial c_{t-1}}\Phi_c(c_{t-1}) + o_{t-1} \odot \Phi_c'(c_{t-1})\right)\\ &= \Phi_g'(W_{xh,i}x_t + W_{hh,i}h_{t-1}) \odot o_{t-1} \odot \Phi_c'(c_{t-1})\end{aligned} \tag{B.16}$$

Here the first factor of the multiplication is bound between 0 and 0.25 and the two other parts are bound between 0 and 1. Therefore, the entire product is bound between 0 and 1. The derivative of the forget gate expands to the same expression, only based on different weight matrices and therefore the same conclusion applies.

Next we do the same for the derivative of the cell candidate.

$$\begin{aligned}\frac{\partial g_t}{\partial c_{t-1}} &= \frac{\partial}{\partial c_{t-1}}\Phi_c(W_{xh,g}x_t + W_{hh,g}h_{t-1})\\ &= \Phi_c'(W_{xh,g}x_t + W_{hh,g}h_{t-1}) \odot \frac{\partial}{\partial c_{t-1}}(W_{xh,g}x_t + W_{hh,g}h_{t-1})\\ &= \Phi_c'(W_{xh,g}x_t + W_{hh,g}h_{t-1}) \odot \left(\frac{\partial o_{t-1}}{\partial c_{t-1}}\Phi_c(c_{t-1}) + o_{t-1} \odot \Phi_c'(c_{t-1})\right)\\ &= \Phi_c'(W_{xh,i}x_t + W_{hh,i}h_{t-1}) \odot o_{t-1} \odot \Phi_c'(c_{t-1})\end{aligned} \tag{B.17}$$

Here we find that the product is bound between 0 and 1. Because both of those terms include the derivative, either of the hyperbolic tangent or the sigmoid function, we can approximate them as zero, since these derivatives are close to zero if the input value is not close to zero. We can therefore approximate (B.15) as the following.

$$\frac{\partial c_t}{\partial c_{t-1}} \approx f_t \tag{B.18}$$

If we insert this result into (B.11), we get the following final expression.

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial y_T}\frac{\partial y_T}{\partial h_T}\frac{\partial h_T}{\partial c_T}\left(\prod_{t=2}^{T} f_t\right)\frac{\partial c_1}{\partial W} \tag{B.19}$$

Here we can see two things. Firstly, since f_t is calculated with the sigmoid function, its value can never be above one and with the terms we approximated to zero it is therefore highly unlikely to result in an exploding gradient. Secondly, we got rid of the derivative of the hyperbolic tangent function which caused the vanishing gradients in the RNN.

In addition to that, in (B.4) we have the term W_{hh}^{T-1}. Since the values in this matrix are constant, we multiply the same values many times, which tends towards zero or ±∞. We did also get rid of this, and in (B.19) we are only multiplying different values of the forget gate, which vary over time and therefore tend less strongly into one direction.

B.2 Code Excerpts

In the following, excerpts of the MATLAB code used to conduct the experiments are displayed.

Code Listing B.1: File `+Functions.signalMagnitudeArea.m`. Implementation of the signal magnitude area.

```matlab
function sma = signalMagnitudeArea(timeBuffer, dataBuffer)
%SIGNALMAGNITUDEAREA Calculate the signal magnitude area of a data buffer
duration = timeBuffer(end) - timeBuffer(1);
sma = 1 / duration * trapz(timeBuffer, sum(abs(dataBuffer - dataBuffer(1,:)).^2, 2));
end
```

Code Listing B.2: File `+Functions.fftPower.m`. Calculation of the frequency power spectrum.

```matlab
function [fftVal, freqs] = fftPower(dataBuffer, sampleRate)
%FFTPOWER Get one sided FFT power spectrum
bufferSize = length(dataBuffer);
fftBothVal = sum(abs(fft(dataBuffer)).^2, 2) ./ bufferSize;
fftVal = fftBothVal(1:floor(bufferSize/2), :);
fftVal(2:end-1, :) = 2 * fftVal(2:end-1, :);
fftVal = fftVal(2:end, :);

freqs = sampleRate * (1:(size(fftBothVal, 1)-2)/2) / (size(fftBothVal, 1) - 2);
end
```

Code Listing B.3: File `+Functions.spectralEntopy.m`. Implementation of the spectral entropy.

```matlab
function H = spectralEntropy(fftPow)
%SPECTRALENTROPY Calculate the spectral entropy from the FFT power
pi1 = zeros(size(fftPow));
pi1(:,sum(fftPow, 1) ~= 0) = fftPow(:, sum(fftPow, 1) ~= 0) ./ sum(fftPow(:, sum(
    fftPow, 1) ~= 0), 1);
H = sum(pi1 .* (pi1 ~= 0) .* log2(pi1 .* (pi1 ~= 0)), 'omitnan');
end
```

Code Listing B.4: File `+CustomNNLayers/+Output/+Relaxed/CrossEntropy.m`. Modification to the loss calculation in the classification layer for the relaxed truth training.

```matlab
classdef CrossEntropy < nnet.internal.cnn.layer.CrossEntropy
    % CrossEntropy   Cross entropy loss output layer

    %   Copyright 2015-2018 The MathWorks, Inc.

    properties
        relaxLength;
    end

    methods
        function this = CrossEntropy(name, numClasses, relaxLength)
            % Output  Constructor for the layer
            % creates an output layer with the following parameters:
            %
            %   name                - Name for the layer
            %   numClasses          - Number of classes. [] if it has to be
            %                       determined later
            this = this@nnet.internal.cnn.layer.CrossEntropy(name, numClasses);
            this.relaxLength = relaxLength;
        end

        function loss = forwardLoss( this, Y, T )
            % forwardLoss    Return the cross entropy loss between estimate
            % and true responses averaged by the number of observations
            %
            % Syntax:
            %   loss = layer.forwardLoss( Y, T );
            %
            % Image Inputs:
            %   Y   Predictions made by network, 1-by-1-by-numClasses-by-numObs
            %   T   Targets (actual values), 1-by-1-by-numClasses-by-numObs
            %
            % Vector Inputs:
            %   Y   Predictions made by network, numClasses-by-numObs-by-seqLength
            %   T   Targets (actual values),  numClasses-by-numObs-by-seqLength

            T = Functions.relaxTruth(T, this.relaxLength);
            loss = forwardLoss@nnet.internal.cnn.layer.CrossEntropy(this, Y, T);
        end

        function dX = backwardLoss( this, Y, T )
            % backwardLoss    Back propagate the derivative of the loss
            % function
            %
            % Syntax:
            %   dX = layer.backwardLoss( Y, T );
            %
            % Image Inputs:
            %   Y   Predictions made by network, 1-by-1-by-numClasses-by-numObs
            %   T   Targets (actual values), 1-by-1-by-numClasses-by-numObs
            %
            % Vector Inputs:
            %   Y   Predictions made by network,  numClasses-by-numObs-by-seqLength
            %   T   Targets (actual values),  numClasses-by-numObs-by-seqLength

            T = Functions.relaxTruth(T, this.relaxLength);
            dX = backwardLoss@nnet.internal.cnn.layer.CrossEntropy(this, Y, T);
        end
    end

    methods (Static)
        function layer = constructWithObservationDim( name, numClasses, categories,
    ...
                observationDim, relaxLength )
            % constructWithObservationDim   Construct a cross entropy layer
            % with the observation dimension and categories defined on
            % construction
            import CustomNNLayers.Output.Relaxed.CrossEntropy
            layer = CrossEntropy( name, numClasses, relaxLength );
            layer.Categories = categories;
        end
    end

end
```

Code Listing B.5: File `+Functions/relaxedTruth.m`. Implementation of the conversion from strict to relaxed truth.

```
function relaxed = relaxTruth(truth, length)
%RELAXTRUTH Relax the ground truth for sequences with one lable that is
%true throughout the whole sequence
labels = truth(:, :, 1);
numLabels = size(labels, 1);

t = -6:12/length:12/length*size(truth, 3)-6;
t = t(1:size(truth, 3));
t = reshape(t, [1, 1, size(truth, 3)]);
t = repmat(t, 1, size(truth, 2));

relaxed = (1 / numLabels + labels * (numLabels - 2) / numLabels) ...
    .* Functions.sigmoid((-1 + labels * 2) .* t) ...
    + labels * 1 / numLabels;
end
```

Code Listing B.6: File `+Functions.accuracy.m`. Implementation of the accuracy measure.

```
function acc = accuracy(groundTruth, prediction)
%ACCURACY Get the accuracy of a prediction
%  groundTruth: A categorical array of the ground truth of a number of
%  samples
%  prediction: A categorial array of the predictions for the samples
if iscell(prediction) && iscell(groundTruth)
    acc = cellfun(@Functions.accuracy, groundTruth, prediction);
    len = cellfun(@length, groundTruth);
    acc = sum(acc .* len) / sum(len);
else
    if all(size(prediction) ~= size(groundTruth)) && numel(prediction) == numel(
    groundTruth)
        prediction = reshape(prediction, size(groundTruth));
    end
    acc = sum(prediction == groundTruth) / numel(groundTruth);
end
end
```

Code Listing B.7: File `+Functions/f1Measure.m`. Implementation of the weighted and non-weighted F-measure.

```
function fm = f1Measure(groundTruth, prediction, measType)
%F1MEASURE The weighted F Measure of the predictions
%  groundTruth: A categorical array of the ground truth of a number of
%  samples
%  prediction: A categorial array of the predictions for the samples
%  type: either 'normal' or 'weighted'

% Default type is 'normal'
if ~exist('measType', 'var')
    measType = 'normal';
end

```

```
if iscell(prediction) && iscell(groundTruth)
    fm = cellfun(@(x,y) Functions.f1Measure(x, y, measType), groundTruth, prediction);
    len = cellfun(@length, groundTruth);
    fm = sum(fm .* len) / sum(len);
else
    if iscolumn(groundTruth) ~= iscolumn(prediction)
        prediction = prediction';
    end

    classes = categories(groundTruth(1));
    fm = 0;
    N = length(groundTruth);
    for i = 1:length(classes)
        if strcmp(measType, 'weighted')
            w = length(groundTruth(groundTruth == classes{i})) / N;
        else
            w = 1/length(classes);
        end
        prc = precision(groundTruth, prediction, classes{i});
        rec = recall(groundTruth, prediction, classes{i});
        f = (prc * rec) / (prc + rec);
        % Handle NaNs as Zeros
        if isnan(f)
            f = 0;
        end
        fm = fm + 2 * w * f;
    end
end
end

function prc = precision(groundTruth, prediction, class)
%PRECISION Get the precision of the prediction for one class
tp = truePositive(groundTruth, prediction, class);
fp = falsePositive(groundTruth, prediction, class);

prc = tp / (tp + fp);
end

function rec = recall(groundTruth, prediction, class)
%RECALL Get the recall of the prediction for one class
tp = truePositive(groundTruth, prediction, class);
fn = falseNegative(groundTruth, prediction, class);

rec = tp / (tp + fn);
end

function tp = truePositive(groundTruth, prediction, class)
%TRUEPOSITIVE Get the number of true positives for a class
trues = groundTruth == class;
tp = sum(prediction(trues) == class);
end

function fp = falsePositive(groundTruth, prediction, class)
%FALSEPOSITIVE Get the number of false positives for a class
positives = prediction == class;
fp = sum(groundTruth(positives) ~= class);
end

function fn = falseNegative(groundTruth, prediction, class)
%FALSENEGATIVE Get the numver of false negatives for a class
trues = groundTruth == class;
fn = sum(prediction(trues) ~= class);
end
```

Code Listing B.8: File `+Networks/+Ha/CNN2D.m`. Reimplementation of the CNN2D architecture from [13].

```
classdef CNN2D < Networks.AbstractHARNetwork
    %CNN2D 2D CNN for activity classification as done in Ha2016

    methods
        function obj = CNN2D(classNum, checkpointPath)
            %CNN2D Construct an instance of this class
            if ~exist('checkpointPath', 'var')
                checkpointPath = pwd;
            end
            obj@Networks.AbstractHARNetwork(checkpointPath);

            obj.layers = [
                % (I1)
                imageInputLayer([17, 60, 1], 'Name', 'I1', 'Normalization', 'none')
                % (C1)
                convolution2dLayer([3, 3], 48, 'Name', 'C1-Conv')
                reluLayer('Name', 'C1-ReLU')
                % (S1)
                maxPooling2dLayer([2, 3], 'Name', 'S1')
                % (C2)
                convolution2dLayer([3, 5], 64, 'Name', 'C2-Conv')
                reluLayer('Name', 'C2-ReLU')
                % (S2)
                maxPooling2dLayer([2, 3], 'Name', 'S2')
                % (F1)
                dropoutLayer(0.5, 'Name', 'F1-Dropout')
                fullyConnectedLayer(512, 'Name', 'F1')
                % (F2)
                dropoutLayer(0.5, 'Name', 'F2-Dropout')
                fullyConnectedLayer(classNum, 'Name', 'F2')
                softmaxLayer('Name', 'F2-Softmax')
                classificationLayer('Name', 'Classification')
                ];

            obj.options = {'sgdm', ...
                'LearnRateSchedule','piecewise',...
                'LearnRateDropPeriod', 2,...
                'LearnRateDropFactor', 0.5,...
                'InitialLearnRate', 0.02,...
                'L2Regularization',0.00005,...
                'MaxEpochs', 30, ...
                'Shuffle', 'every-epoch', ...
                'Verbose', false, ...
                'Plots', 'training-progress'};
        end

    end
end
```

Code Listing B.9: File `+Networks.AbstractHARNetwork.m`. The abstract base class for all out implementations of neural networks.

```
classdef AbstractHARNetwork < handle & matlab.mixin.Copyable
    %ABSTRACTHARNETWORK Abstract base class for all HAR classification
    %networks

    properties
        network
        layers
        options
        checkpointPath
        trainResults = struct();
        performance;
        finalIndex;
        miniBatchSize = 2048;
        predictionMethod = 'raw';
    end

    methods
        function obj = AbstractHARNetwork(checkpointPath)
            obj.checkpointPath = checkpointPath;
        end

        function trainResults = train(obj, Xtrain, Ytrain, Xval, Yval, varargin)
            %TRAIN Train the network
            time = clock;
            checkpointFolder = sprintf('%s\\%d_%d_%d_%d_%d_%d', obj.checkpointPath, ...
                time(1), time(2), time(3), time(4), time(5), time(6));

            mkdir(checkpointFolder);

            trainOptions = obj.getTrainOptions(Xtrain, Xval, Yval, ...
                'CheckpointPath', checkpointFolder, varargin{:});
            if isempty(obj.network)
                [~, trainResults] = trainNetwork(Xtrain, Ytrain, obj.layers, trainOptions);
            else
                [~, trainResults] = trainNetwork(Xtrain, Ytrain, obj.network.Layers, trainOptions);
            end

            % Get best network
            ind = ~isnan(trainResults.ValidationLoss);
            ind(1) = 0;
            meanLoss = (3*trainResults.ValidationLoss(ind) + trainResults.TrainingLoss(ind)) / 4;
            meanAcc = (trainResults.ValidationAccuracy(ind) + trainResults.TrainingAccuracy(ind)) / 2;
            [perf, best] = min(3 * meanLoss * 100 + 100 - meanAcc);
            checkPointNumber = find(ind);
            checkPointNumber = checkPointNumber(best);
            files = dir(checkpointFolder);
            files = files(~ismember({files.name},{'.','..'}));
            tmp = load([files(contains({files.name},['net_checkpoint__' num2str(checkPointNumber) '__'])).folder, '/',...
                files(contains({files.name},['net_checkpoint__' num2str(checkPointNumber) '__'])).name],'net');

            obj.network = tmp.net;
            obj.trainResults = trainResults;
            obj.finalIndex = best;
            obj.performance = perf;
        end

        function options = getTrainOptions(obj, Xtrain, Xval, Yval, varargin)
            opts = obj.options;
            if exist('Xval', 'var') && exist('Yval', 'var')
                opts = [opts, 'ValidationData', {{Xval, Yval}}];
            end

            if iscell(Xtrain)
                dataLength = length(Xtrain);
            else
                dataLength = size(Xtrain, 4);
            end
```

```matlab

            options = trainingOptions(opts{:}, ...
                'MiniBatchSize', obj.miniBatchSize, ...
                'OutputFcn', @(info) Networks.stopIfNotImproving(info, 12), ...
                'ValidationFrequency', floor(dataLength / obj.miniBatchSize), ...
                varargin{:});
        end

        function pred = classify(net, X, varargin)
            % Parse Arguments
            parser = inputParser;
            addRequired(parser, 'net', @(x) isa(x, 'Networks.AbstractHARNetwork'));
            addRequired(parser, 'x');

            parse(parser, net, X);

            net = parser.Results.net;
            X = parser.Results.x;

            pred = classify(net.network, X);

            pred = sequence2label(pred, 'Method', net.predictionMethod, varargin{:});
        end

        function [accuracy, weightedFMeas] = validate(obj, X, Y)
            YPred = obj.classify(X);
            accuracy = Functions.accuracy(Y, YPred);
            weightedFMeas = Functions.f1Measure(Y, YPred, 'weighted');
        end

        function analyze(obj)
            analyzeNetwork(obj.layers);
        end

        function plotConfusion(obj, X, Y, title)
            YPred = obj.classify(X);
            plotconfusion(Y, YPred);
            if exist('title', 'var')
                sgtitle(title);
            end
        end

        function plotTraining(obj, title)
            validationInd = find(~isnan(obj.trainResults.ValidationAccuracy));
            figure;
            ax = subplot(2, 1, 1);
            hold on;
            % Plot accuracy
            plot(obj.trainResults.TrainingAccuracy, 'b');
            plot(validationInd, obj.trainResults.ValidationAccuracy(validationInd), '--ok');
            ylabel('Accuracy / %');
            % Plot loss
            ax(2) = subplot(2, 1, 2);
            hold on;
            plot(obj.trainResults.TrainingLoss, 'Color', [0.8500 0.3250 0.0980]);
            plot(validationInd, obj.trainResults.ValidationLoss(validationInd), '--ok');
            ylabel('Loss');
            xlabel('Iterations');
            hold off;

            linkaxes(ax, 'x');
            if exist('title', 'var')
```

```
                sgtitle(title);
            else
                sgtitle(sprintf('%s: Training Performance', class(obj)));
            end
        end

        function plotSequenceClass(obj, X, Y, title)
            maxPlots = 10;
            YPred = obj.classify(X);
            figure;

            plotInd = 1:length(Y);
            if maxPlots < length(Y)
                rnd = randperm(length(Y));
                plotInd = rnd(1:maxPlots);
            end

            for i = 1:min([length(Y), maxPlots])
                subplot(min([length(Y), maxPlots]), 1, i);
                hold on;
                % Plot ground truth
                plot(1:length(Y{plotInd(i)}), Y{plotInd(i)});
                % Plot prediction
                plot(1:length(YPred{plotInd(i)}), YPred{plotInd(i)});
                % Define legend
                legend('Ground Truth', 'Prediction');
                hold off;
            end

            if exist('title', 'var')
                sgtitle(title);
            else
                sgtitle(sprintf('%s: Prediction vs. Ground Truth', class(obj)));
            end
        end

        function plotValidation(obj, X, Y, title)
            if iscell(Y) && ~isscalar(Y{1}) && strcmp(obj.predictionMethod, 'raw')
                if exist('title', 'var')
                    obj.plotSequenceClass(X, Y, title);
                else
                    obj.plotSequenceClass(X, Y);
                end
            else
                if iscell(Y)
                    Y = sequence2label(Y, 'Method', obj.predictionMethod);
                end
                if exist('title', 'var')
                    obj.plotConfusion(X, Y, title);
                else
                    obj.plotConfusion(X, Y);
                end
            end
        end

        function comp = getComplexity(obj)
            compLayers = obj.network.Layers;
            lstms = arrayfun(@(x) isa(x, 'nnet.cnn.layer.LSTMLayer'), compLayers);
            fcs = arrayfun(@(x) isa(x, 'nnet.cnn.layer.FullyConnectedLayer'),
    compLayers);
            compLayers = compLayers(or(lstms, fcs));

            comp = 0;
            for layer = compLayers'
```

```
                switch class(layer)
                    case 'nnet.cnn.layer.LSTMLayer'
                        comp = comp + 8 * (layer.InputSize + layer.NumHiddenUnits) ...
                            * layer.NumHiddenUnits + 4 * layer.NumHiddenUnits;

                    case 'nnet.cnn.layer.FullyConnectedLayer'
                        comp = comp + 2 * layer.InputSize * layer.OutputSize;
                end
            end
        end
    end
end

function label = sequence2label(sequence, varargin)
parser = inputParser;
validMethod = @(x) strcmp(x, 'raw') || strcmp(x, 'last') || ...
    strcmp(x, 'most') || strcmp(x, 'smooth');
addRequired(parser, 'sequence');
addParameter(parser, 'Method', 'raw', validMethod);
addParameter(parser, 'AverageLength', 0, ...
    @(x) validateattributes(x,{'numeric'}, {'nonempty', 'integer', 'positive', 'scalar
    '}));

parse(parser, sequence, varargin{:});

sequence = parser.Results.sequence;
method = parser.Results.Method;
avgLength = parser.Results.AverageLength;

switch method
    case 'last'
        % Return the last label
        if iscell(sequence)
            label = cellfun(@(x) x(end), sequence);
        end
    case 'most'
        % Return the label that occurs most often
        if iscell(sequence)
            label = cellfun(@(x) mode(x), sequence);
        end
    case 'smooth'
        % Smooth output with a moving average Filter
        if iscell(sequence)
            label = cellfun(@(x) Functions.movingAverage(x, 'Length', avgLength), ...
                sequence, 'UniformOutput', false);
        end
    otherwise
        label = sequence;
end
end
```

Code Listing B.10: File `+Networks.RNNClassifier.RelaxedLSTM.m`. Implementation of the RelaxedLSTM architecture with relaxed truth training.

```
classdef RelaxedLSTM < Networks.AbstractHARNetwork
    %SequenceLSTM LSTM with sequence to sequence classification
    %   for classifying activity

    methods
        function obj = RelaxedLSTM(numHiddenUnits, numFeatures, numClasses,
    checkpointPath)
            %LastLSTM Construct an instance of this class
            if ~exist('checkpointPath', 'var')
                checkpointPath = pwd;
            end
            obj@Networks.AbstractHARNetwork(checkpointPath);

            obj.layers = constructNet(numHiddenUnits, numFeatures, numClasses);

            obj.options = {'adam', ...
                'LearnRateSchedule','piecewise',...
                'LearnRateDropPeriod', 2,...
                'LearnRateDropFactor', 0.5,...
                'InitialLearnRate', 0.12,...
                'L2Regularization', 0.005,...
                'MaxEpochs', 100, ...
                'Shuffle', 'every-epoch', ...
                'Verbose', false, ...
                'Plots', 'training-progress'};

            obj.miniBatchSize = 256;
        end

    end
end

function layers = constructNet(numHiddenUnits, numFeatures, numClasses)
%CONSTRUCTNET Construct the network

layers = [ ...
    sequenceInputLayer(numFeatures, 'Name', 'Input')
    dropoutLayer('Name', 'Dropout')];

for i = 1:length(numHiddenUnits)
    layers = [layers; ...
        lstmLayer(numHiddenUnits(i), 'OutputMode', 'sequence', 'Name', sprintf('LSTM%d', i))];
end

layers = [layers; ...
    fullyConnectedLayer(numClasses, 'Name', 'FC')
    softmaxLayer('Name', 'Softmax')
    CustomNNLayers.relaxedClassificationLayer('Name', 'relaxedClassification')];
end
```

Code Listing B.11: File `+Networks.stopIfNotImproving.m`. Function to determine if the training of a network should be stopped early.

```
function stop = stopIfNotImproving(info, N)

stop = false;

% Keep track of the best validation accuracy and the number of validations for which
% there has not been an improvement of the accuracy.
persistent bestValAccuracy
persistent valLag

% Clear the variables when training starts.
if info.State == "start"
    bestValAccuracy = 1e5;
    valLag = 0;

elseif ~isempty(info.ValidationLoss)

    % Compare the current validation accuracy to the best accuracy so far,
    % and either set the best accuracy to the current accuracy, or increase
    % the number of validations for which there has not been an improvement.


    meanLoss = (3*info.ValidationLoss + info.TrainingLoss)/4;
    meanAcc = (info.ValidationAccuracy + info.TrainingAccuracy)/2;
    best = 3*meanLoss*100+100-meanAcc;

    if bestValAccuracy > best
        valLag = 0;
        bestValAccuracy = best;
    else
        valLag = valLag + 1;
    end

    % If the validation lag is at least N, that is, the validation accuracy
    % has not improved for at least N validations, then return true and
    % stop training.
    if valLag >= N
        stop = true;
    end

end

end
```

C Appendix for Chapter 5

In this appendix, additional results of the experiments can be found.

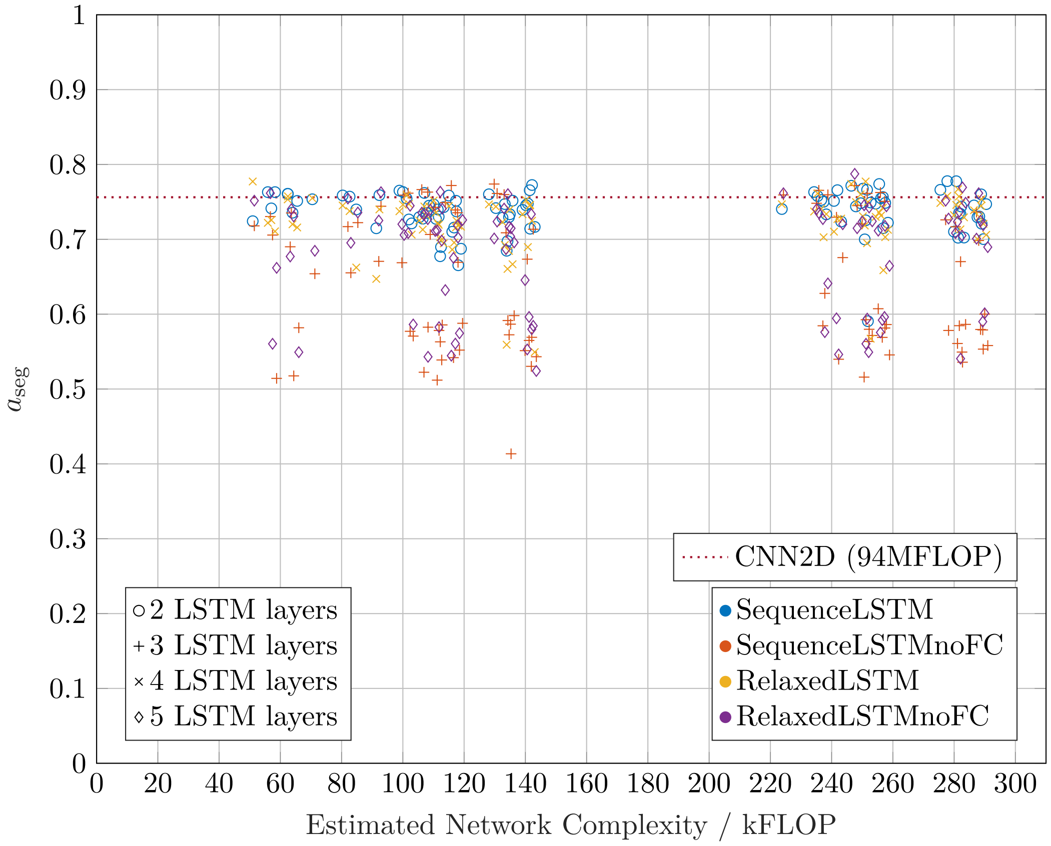


Figure C.1: Accuracy on segmented data over network complexity for different architectures.

Table C.1: Performance results for different groups of networks in the architecture study.

ARCHITECTURES		PERFORMANCE MEASURE ($\mu(s)$)							
Basic	*No. LSTM-layers*	$a_{\text{seg}}/\%$		$a_{\text{seq}}/\%$		$F_{1,\text{seg}}/\%$		$F_{1,\text{seq}}/\%$	
All	All	68.1	(7.3)	54.2	(7.8)	72.1	(7.9)	47.6	(10.6)
	2	74.7	(2.3)	64.3	(3.4)	79.3	(2.4)	61.3	(5.4)
	3	69.0	(5.9)	57.5	(6.6)	73.2	(6.3)	52.8	(8.7)
	4	65.7	(8.1)	51.7	(7.4)	69.2	(8.7)	43.7	(9.8)
	5	66.8	(7.5)	49.5	(5.3)	70.6	(8.2)	40.9	(6.2)
SequenceLSTM	All	73.8	(2.8)	62.7	(6.4)	78.1	(2.8)	60.5	(7.8)
	2	75.8	(1.3)	64.8	(5.1)	80.0	(1.1)	62.7	(5.9)
	3	74.7	(1.5)	65.7	(2.6)	79.0	(1.3)	64.2	(2.5)
	4	74.1	(2.1)	64.1	(3.7)	78.7	(1.8)	62.4	(4.3)
	5	72.0	(3.5)	58.5	(8.3)	76.2	(3.8)	55.1	(10.4)
SequenceLSTMnoFC	All	64.3	(9.0)	50.3	(8.6)	59.9	(21.5)	41.2	(11.6)
	2	73.8	(2.9)	61.4	(3.5)	73.0	(19.3)	57.7	(5.5)
	3	68.4	(8.2)	54.5	(7.0)	67.2	(17.7)	47.0	(9.3)
	4	62.5	(8.9)	48.7	(8.1)	59.0	(20.6)	39.0	(9.2)
	5	58.5	(6.2)	43.8	(4.3)	49.5	(21.7)	31.6	(5.0)
RelaxedLSTM	All	72.2	(4.0)	62.2	(5.8)	77.2	(3.7)	59.6	(7.6)
	2	74.8	(1.5)	65.1	(2.7)	79.7	(1.5)	63.4	(3.0)
	3	73.3	(2.7)	64.8	(3.1)	78.1	(2.5)	63.0	(3.4)
	4	72.6	(2.6)	63.9	(3.8)	77.6	(2.3)	62.1	(4.5)
	5	70.1	(5.3)	57.6	(6.9)	75.1	(5.0)	53.5	(9.3)
RelaxedLSTMnoFC	All	68.1	(7.3)	54.2	(7.8)	72.1	(7.9)	47.6	(10.6)
	2	74.7	(2.3)	64.3	(3.4)	79.3	(2.4)	61.3	(5.4)
	3	69.0	(5.9)	57.5	(6.6)	73.2	(6.3)	52.8	(8.7)
	4	65.7	(8.1)	51.7	(7.4)	69.2	(8.7)	43.7	(9.8)
	5	66.8	(7.5)	49.5	(5.3)	70.6	(8.2)	40.9	(6.2)

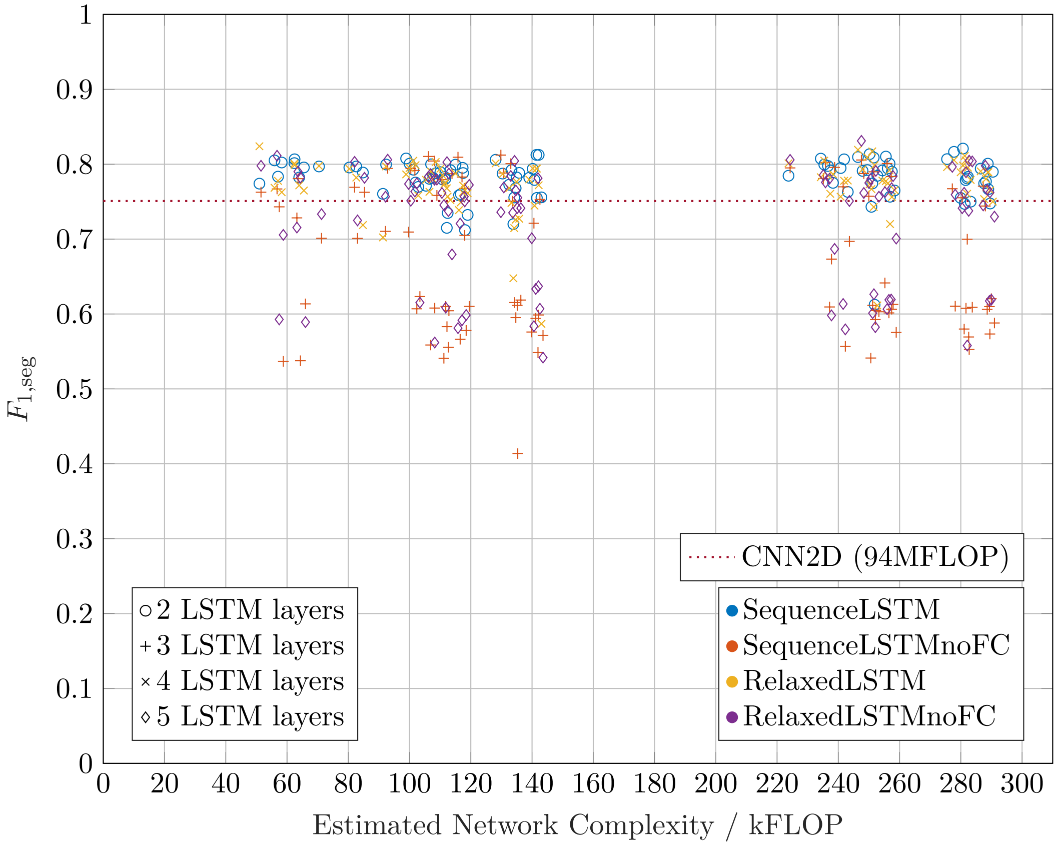


Figure C.2: F-measure on segmented data over network complexity for different architectures.

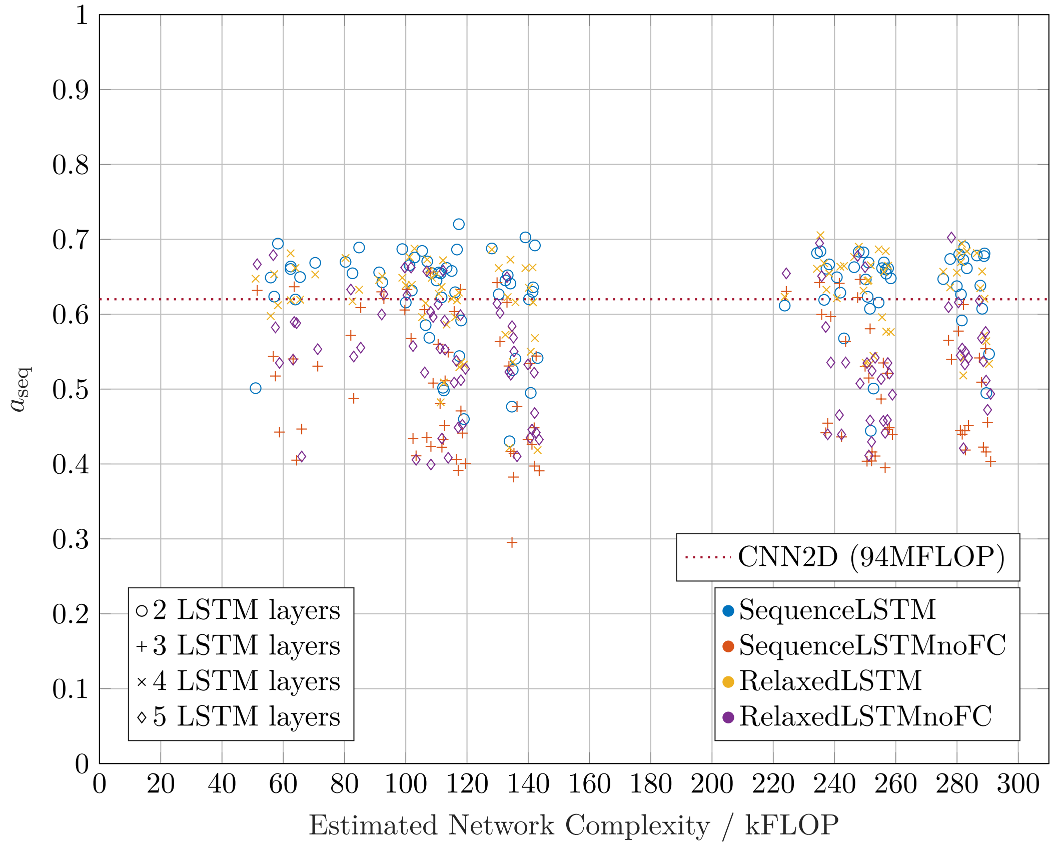


Figure C.3: Accuracy on sequence data over network complexity for different architectures.

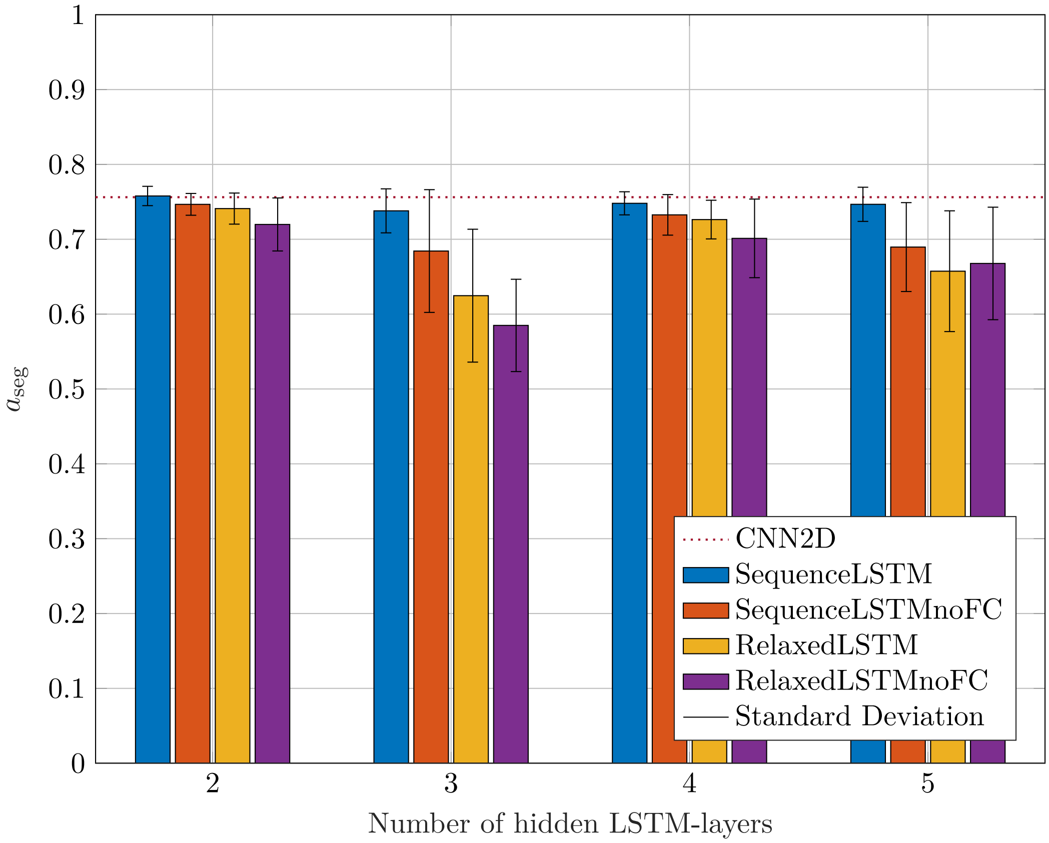


Figure C.4: Accuracy on segmented data over network depth for different architectures.

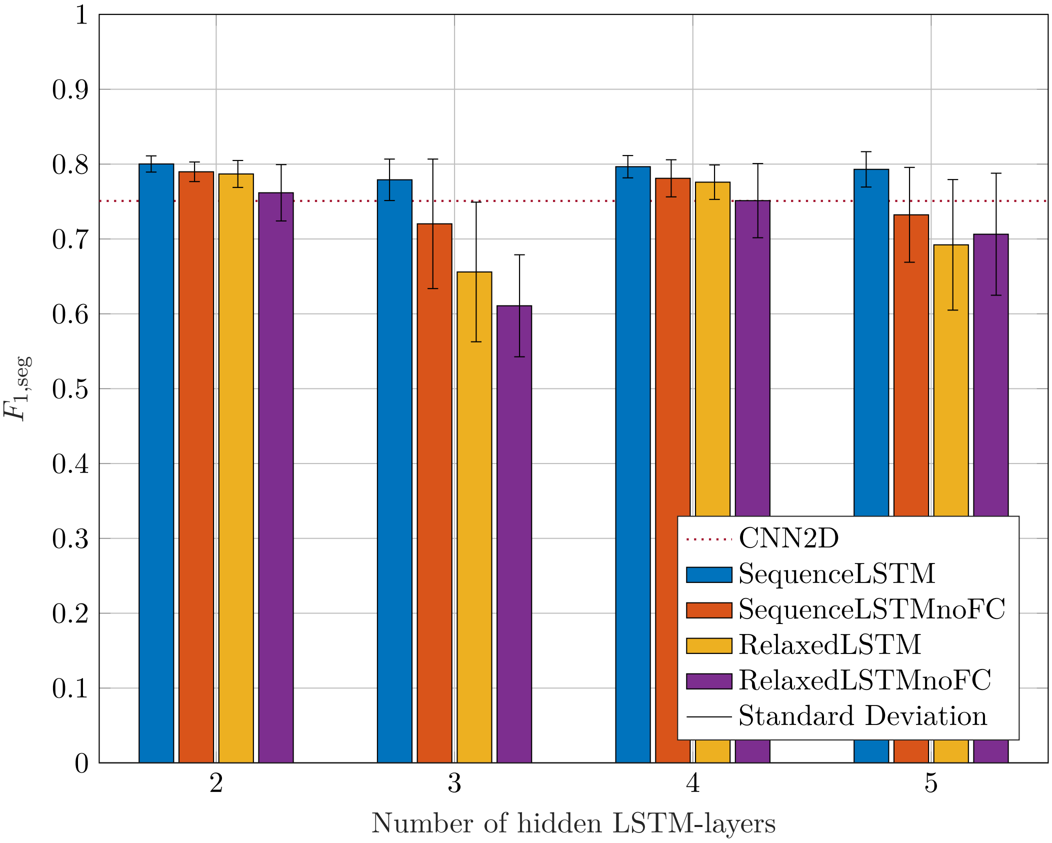


Figure C.5: F-measure on segmented data over network depth for different architectures.

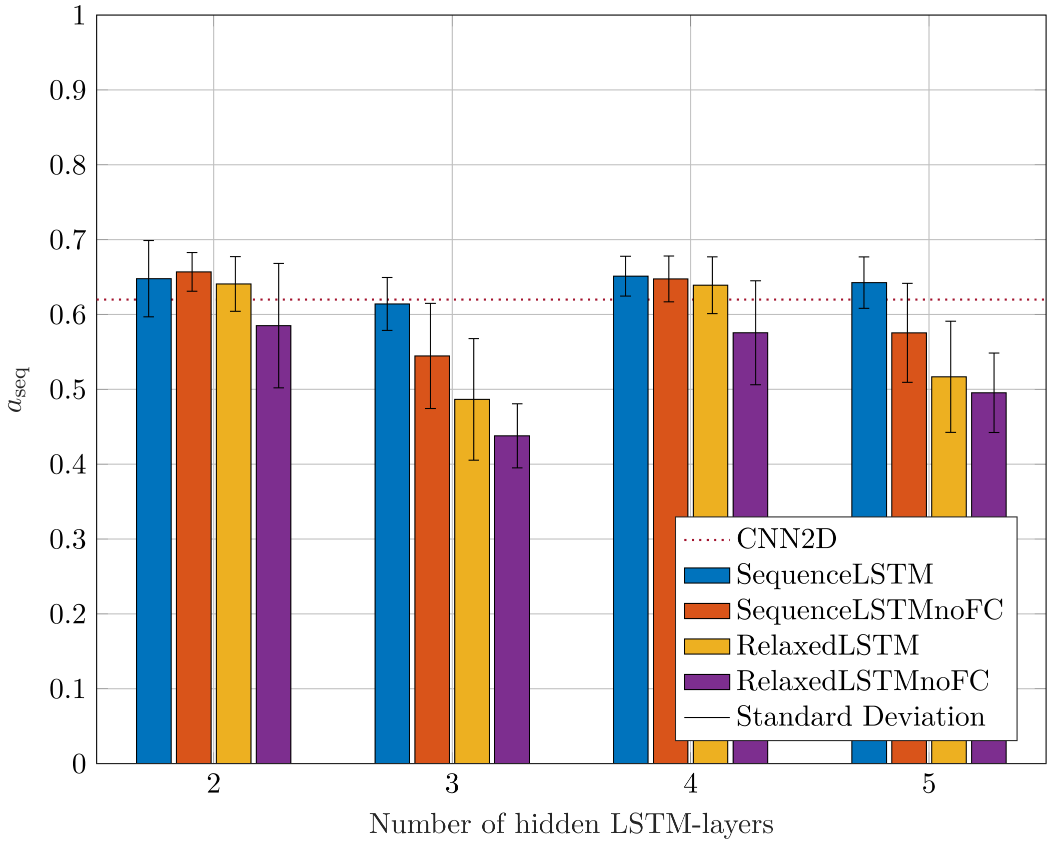
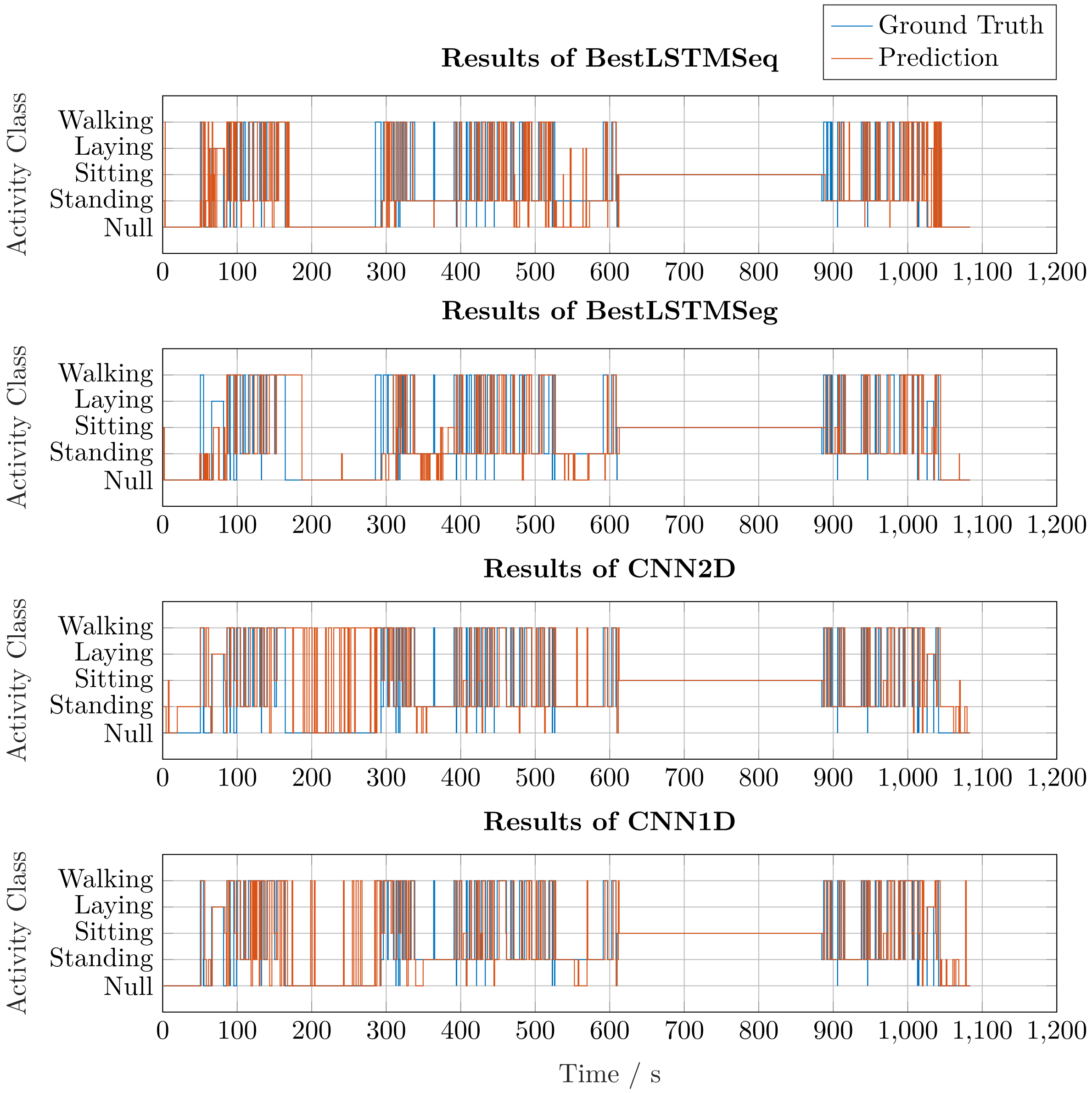


Figure C.6: Accuracy on sequence data over network depth for different architectures.



Results of BestLSTMSeq

Results of BestLSTMSeg

Results of CNN2D

Results of CNN1D

Figure C.7: Continuous classification progression of two LSTMs and two CNNs on the Opportunity dataset.

Table C.2: Performance comparison of our approaches with all entries to the Opportunity challenge. Additional information about the individual approaches should be taken from [6].

NETWORK/METHOD	PERFORMANCE
	$F_{1,\mathrm{seq}}/\%$
BestLSTMSeq	70
BestLSTMSeg	67
CNN2D	62
CNN1D	65
LDA	64
QDA	66
NCC	58
1 NN	85
3 NN	86
UP	58
NStar	58
SStar	61
CStar	60
NU	54
MI	85
MU	57
UT	48